UNIVERSIDADE FEDERAL DO PAMPA
Ciência da Computação

TONY FARNEY BRUCK MENDES RIBEIRO

DETECÇÃO DE FALHAS E RE-ROTEAMENTO AUTOMÁTICO E TRANSPARENTE DE PACOTES PARA SERVIÇOS DE REDE COM REDUNDÂNCIA IMPLEMENTADA

Trabalho de Conclusão de Curso

Alegrete
2012

TONY FARNEY BRUCK MENDES RIBEIRO

DETECÇÃO DE FALHAS E RE-ROTEAMENTO AUTOMÁTICO E TRANSPARENTE DE PACOTES PARA SERVIÇOS DE REDE COM REDUNDÂNCIA IMPLEMENTADA

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Orientador: Diego Luis Kreutz

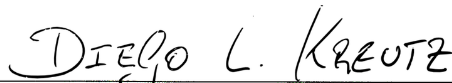
Co-orientador: Douglas Dyllon Jeronimo de Macedo

TONY FARNEY BRUCK MENDES RIBEIRO

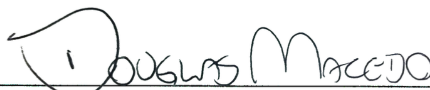
DETECÇÃO DE FALHAS E RE-ROTEAMENTO AUTOMÁTICO E TRANSPARENTE DE PACOTES PARA SERVIÇOS DE REDE COM REDUNDÂNCIA IMPLEMENTADA

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

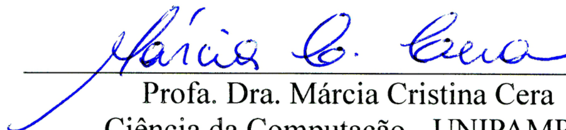
Trabalho apresentado e aprovado em: 06 de Janeiro de 2012.
Banca Examinadora:



Prof. Msc. Diego Luis Kreutz
Orientador
Ciência da Computação - UNIPAMPA



Msc. Douglas Dyllon Jeronimo de Macedo
Coorientador
Laboratório de Pesquisa em Sistemas Distribuídos - UFSC



Profa. Dra. Márcia Cristina Cera
Ciência da Computação - UNIPAMPA

Ademir Dias Lima
Núcleo de Tecnologia da Informação e Comunicação - UNIPAMPA

Este trabalho é dedicado a todos aqueles que direta ou indiretamente contribuiram na minha formação.

Agradecimentos

Agradeço aos meus pais, Maria Solange e Moacir, por tudo que fizerem para que conseguisse chegar até aqui. Obrigado por terem sido o ombro amigo no qual sempre pude me apoiar nos momentos de dificuldade. Sempre estiveram do meu lado, orientando e apoiando minhas decisões. Se não fosse por vocês, talvez nem tivesse começado a faculdade.

Agradeço a todos os professores, por promoverem o conhecimento e transformar alunos em profissionais. Em especial, agradeço ao prof. Diego Luis Kreutz, meu professor desde algoritmos até sistemas distribuídos, orientador durante o período que fui bolsista do NTIC, e orientador do meu trabalho de conclusão. Foi com quem aprendi a programar e a ser um bom programador. Obrigado por todo o apoio, ensinamento, por ter acreditado no meu potencial, por ter ajudado a me tornar autodidata, enfim, por ter me mostrado o caminho para o profissionalismo.

Agradeço àqueles colegas que me apoiaram e acabaram por se tornar grandes amigos. Em especial, agradeço ao Madson, que desde de o começo foi um grande amigo. Posso dizer que foi quem me ajudou a dar os primeiros passos, pois quando entrei na universidade, mal sabia ligar o computador. Agradeço em especial também ao Adriano e ao Marcelo, com quem tive o prazer de compartilhar idéias, estudar, resolver problemas e trabalhar. Não poderia esquecer o William, amigo para toda a hora, companheiro de festas e com quem tive a oportunidade de conversar sobre assuntos dos mais diversos.

Agradeço ao Cristian Tales pelo apoio e compreensão durante o período que trabalhei com ele.

Por fim, agradeço a todos aqueles que direta ou indiretamente de alguma maneira ajudaram na minha formação e desenvolvimento deste trabalho.

Obrigado!

Resumo

A replicação é uma das técnicas utilizadas com frequência para aumentar a disponibilidade de serviços de rede. Através da replicação, obtém-se a redundância, isto é, a repetição de componentes. Quando bem implementada, pode evitar que os clientes, sejam eles usuários ou sistemas, fiquem sem acesso aos serviços por períodos transitórios ou mesmo permanentes.

O objetivo do trabalho é propor uma solução, em *software*, que proporcione maiores índices de disponibilidade em ambientes que possuam redundância implementada em serviços de rede essenciais, como DNS, LDAP e SMTP. Os clientes que utilizam os serviços de rede normalmente possuem configurações estáticas ou apontam para uma única réplica. Neste sentido, a meta do trabalho é aumentar o índice de disponibilidade através de uma melhor utilização das réplicas, de forma transparente e automática aos clientes. Para atingir este resultado, a meta é utilizar os recursos disponibilizados pelos *firewalls*, considerados elementos essenciais e normalmente presentes nas redes locais.

Este trabalho apresenta uma solução, denominada *AvailableD*, para aumentar a disponibilidade de serviços de rede com redundância implementada em ambientes que possuem *firewalls* GNU/Linux em operação. A solução proposta é transparente para os clientes dos serviços. Além disso, visa a simplicidade e facilidade de uso e aplicação. O *AvailableD* é composto por três tipos de elementos básicos: a) os dispositivos de monitoramento de serviços; b) os mecanismos de re-roteamento de pacotes; e c) os dispositivos para envio de notificações. O primeiro passo é detectar a falha em alguma das réplicas dos serviços. O segundo passo é aplicar políticas de re-roteamento dos pacotes, para que os pedidos das réplicas falhas sejam redirecionados para réplicas ativas. Por fim, as falhas nas réplicas são reportadas aos administradores da rede.

O *AvailableD* foi implementado como uma ferramenta *open source*, extensível e adaptável a diferentes situações, o que é atingido através de uma arquitetura baseada em *plugins* que executam tarefas como monitoramento dos serviços, a adição das regras de re-roteamento em *firewalls* e o envio de notificações. A solução foi testada e validada em um cenário baseado no contexto de serviços do NTIC da UNIPAMPA. Os resultados permitiram concluir que os objetivos propostos foram atingidos, ou seja, a ferramenta torna possível um aumento transparente e automático, para os clientes, nos índices de disponibilidade de serviços de rede redundantes.

Palavras-chave: disponibilidade, serviços de rede, redundância, firewall, re-roteamento.

Abstract

Replication is frequently used to increase the availability of network services. Redundancy is the direct result of component or service replication. Replication, when properly implemented, can prevent clients (users or systems) to keep without access to services during temporary or permanent periods.

The goal of this work is to propose a software solution that is capable of providing higher levels of availability in environments that have redundancy implemented in essential network services such as DNS, LDAP and SMTP. Clients who use the network services typically have static configurations or support only a single address, to one replica. Having this in mind, the goal of the work is to increase the availability rate, from the client perspective, by applying transparent and automated techniques, such as packet re-routing. To achieve this result, the idea is to use resources and tools available inside firewalls, which are important network components usually present in local networks.

This work presents the solution called AvailableD. It automatically and transparently increases the availability of network services with implemented redundancy. To do so, it uses resources within GNU/Linux firewalls. The solution was designed with simplicity and ease of use in mind. It consists of three basic elements: a) plug-ins for services monitoring; b) packet re-routing mechanisms; c) resources for sending notifications to network administrators. The first step is to detect a failure of any service replica. The second step is to apply packet re-routing rules, so that the requests addressed to failed replicas are redirected to active replicas. Finally, failures in replicas are reported to network administrators.

The AvailableD is implemented as an open source, extensible and adaptable tool. The adaptability is achieved through an architecture which is based on plug-ins that perform tasks such as services monitoring, re-routing rules management and notifications delivery. The solution has been tested and evaluated in a scenario similar to the NTIC's, of UNIPAMPA, environment. The results show that the main goal of the work was reached through the transparent and automated increase of the availability rate for network services with implemented replication.

Keywords: availability, network services, redundancy, firewall, re-routing.

LISTA DE FIGURAS

3.1 Ambiente alvo de aplicação da nova solução proposta . . . . p. 15

3.2 Legenda para fluxogramas de execução . . . . p. 16

3.3 Funcionamento simplificado da solução *AvailableD* . . . . p. 17

3.4 *AvailableD* monitorando o serviço HTTP em dois servidores . . . . p. 17

3.5 Redirecionamento realizado pelo *AvailableD* após identificação de falha . . . p. 18

3.6 *AvailableD* removendo redirecionamento após o retorno a disponibilidade . . p. 18

3.7 Requisição HTTP para um servidor disponível . . . . p. 19

3.8 Requisição HTTP para um servidor indisponível . . . . p. 19

3.9 Arquitetura do *AvailableD*, seus *plugins* e programas externos . . . . p. 23

3.10 Sintaxe básica do arquivo de configuração do *AvailableD* . . . . p. 25

3.11 Exemplo de configuração com sobrescrita de valores dos parâmetros . . . . . p. 26

3.12 Fluxograma de execução do módulo configuração do *AvailableD* . . . . p. 27

3.13 Funcionamento genérico dos *plugins* verificadores . . . . p. 29

3.14 Funcionamento do módulo monitoramento do *AvailableD* . . . . p. 30

3.15 Funcionamento do módulo executor de ações do *AvailableD* . . . . p. 31

3.16 Funcionamento do módulo log do *AvailableD* . . . . p. 32

3.17 Funcionamento do módulo notificação do *AvailableD* . . . . p. 33

3.18 Lógica de integração entre os módulos do *AvailableD* . . . . p. 34

4.1 Algumas das expressões regulares utilizadas na análise léxica . . . . p. 37

4.2 Condições de início utilizadas para a análise sintática . . . . p. 38

4.3 Código que processa a palavra reservada “service” . . . . p. 38

4.4 Código que interpreta as variáveis do módulo monitoramento . . . . p. 40

4.5 Trecho do código do *plugin* smtp.monitor . . . . p. 42

4.6 Função que trata redirecionamentos para servidores indisponíveis . . . . . . p. 44

4.7 Condição de corrida do módulo executor de ações . . . . p. 45
4.8 Código do gerenciador de *log* que escreve em arquivos . . . . p. 46
4.9 Trecho de um arquivo de *log* gerado pelo gerenciador implementado . . . . . p. 46
4.10 Trecho da função responsável por realizar as notificações . . . . p. 47
4.11 Código que cria os *threads* de monitoramento . . . . p. 48
4.12 Trecho de código dos *threads* de monitoramento . . . . p. 49
5.1 Ambiente utilizado para testes do *AvailableD* . . . . p. 54
5.2 Utilização média da UCP em 300 segundos de monitoramento . . . . p. 61
5.3 Alocação média de memória virtual em 300 segundos de monitoramento . . . p. 61
5.4 Tráfego de rede médio gerado em 300 segundos de monitoramento . . . . . . p. 62
5.5 Utilização da UCP para um intervalo entre verificações de 15 segundos . . . . p. 63
5.6 Duração média dos períodos de indisponibilidade percebidos pelos clientes . p. 64
5.7 Comparação dos períodos de indisponibilidade entre os serviços . . . . p. 64
A.1 Arquitetura do Linux-HA . . . . p. 72
D.1 LVS com NAT. Figura adaptada de (LVS, 2011) . . . . p. 80
D.2 LVS com Roteamento Direto. Figura adaptada de (LVS, 2011) . . . . p. 81
D.3 LVS com Tunelamento IP. Figura adaptada de (LVS, 2011) . . . . p. 83
E.1 Arquitetura do Ultra Monkey . . . . p. 84
H.1 Código do *script* para coleta da utilização da UCP . . . . p. 147
H.2 Código do *script* para coleta da memória virtual alocada . . . . p. 148
H.3 Código do *script* para coleta do tráfego de rede gerado . . . . p. 149
H.4 Código *script* para simulação de falhas em serviços . . . . p. 150
H.5 *Script* PERL que abre conexão via *socket* . . . . p. 151
H.6 *Script* para coleta dos *outages* dos serviços HTTP e SMTP . . . . p. 152
H.7 *Script* para coleta dos *outages* do serviço DNS . . . . p. 153
I.1 Configuração do AvailableD para monitorar 6 réplicas . . . . p. 155

LISTA DE TABELAS

2.1 Síntese das características analisadas no Linux-HA, LVS e Ultra Monkey . . p. 12

5.1 Comparação entre o AvailableD, Linux-HA, LVS e Ultra Monkey . . . . . . p. 66

LISTA DE SIGLAS E ABREVIATURAS

ARP	*Address Resolution Protocol*
CSV	*Comma-Separated Values*
DNS	*Domain Name Server*
DRBD	*Distributed Replicated Block Device*
GPLv3	*Gnu General Public License version 3*
HTTP	*Hypertext Transfer Protocol*
HTML	*Hypertext Markup Language*
IP	*Internet Protocol*
LDAP	*Lightweight Directory Access Protocol*
Linux-HA	*Linux High Availability*
LVS	*Linux Virtual Server*
Mbps	Megabits por segundo
MV	*Máquina Virtual*
NAT	*Network Address Translation*
NRPE	*Nagios Remote Plugin Executor*
NTIC	Núcleo de Tecnologia da Informação e Comunicação
SMTP	*Simple Mail Transfer Protocol*
SO	Sistema Operacional
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
TIC	Tecnologia da Informação e Comunicação
TLS	*Transport Layer Security*
UCP	Unidade Central de Processamento
UDP	*User Datagram Protocol*
UNIPAMPA	Universidade Federal do Pampa

SUMÁRIO

1 Introdução p. 1

2 Revisão Bibliográfica p. 3

2.1 Linux-HA . . . . p. 3

2.2 Linux Virtual Server (LVS) . . . . p. 4

2.3 Ultra Monkey . . . . p. 6

2.4 Nagios . . . . p. 7

2.5 Zabbix . . . . p. 9

2.6 Análise das soluções . . . . p. 10

3 A Solução Proposta p. 13

3.1 Visão Geral . . . . p. 14

3.2 Premissas e Requisitos . . . . p. 19

3.3 Arquitetura do AvailableD . . . . p. 21

3.3.1 Módulo Configuração . . . . p. 22

3.3.2 Módulo Monitoramento . . . . p. 27

3.3.3 Módulo Executor de Ações . . . . p. 29

3.3.4 Módulo Log . . . . p. 30

3.3.5 Módulo Notificação . . . . p. 32

3.3.6 Módulo Integrador . . . . p. 33

4 Tecnologias e Implementação p. 35

4.1 Tecnologias de Apoio . . . . p. 35

4.2 Arquitetura Concretizada na Implementação . . . . p. 37

4.2.1 Configuração . . . . p. 37

4.2.2 Monitoramento de Serviços . . . . p. 38
4.2.3 Executor de Ações de Resposta . . . . p. 41
4.2.4 Log . . . . p. 44
4.2.5 Envio de Notificações . . . . p. 46
4.2.6 Integração . . . . p. 47
4.2.7 Instalação . . . . p. 48
4.2.8 Operação . . . . p. 50

5 Testes e Resultados p. 52
5.1 Ambiente de Testes . . . . p. 52
5.2 Testes Funcionais . . . . p. 53
5.3 Testes de Desempenho . . . . p. 55
5.4 Resultados Obtidos . . . . p. 60
5.5 Comparação com Outras Soluções . . . . p. 64

6 Considerações Finais p. 67

Referências Bibliográficas p. 69

Apêndice A – Arquitetura do Linux-HA p. 71

Apêndice B – Instalação, funcionamento e configuração do Linux-HA p. 74
Configuração do Heartbeat . . . . p. 74
Configuração do Pacemaker . . . . p. 75

Apêndice C – Algoritmos de escalonamento do LVS p. 77

Apêndice D – Métodos de roteamento suportados pelo LVS p. 79
NAT . . . . p. 79
Roteamento direto . . . . p. 80

Tunelamento IP . . . . p. 82

Apêndice E – Arquitetura do Ultramonkey p. 84

Apêndice F – Documento de Requisitos p. 85

Apêndice G – Documento de Projeto p. 97

Apêndice H – Scripts utilizados para os testes p. 147

Apêndice I – Exemplo de configuração do AvailableD utilizada nos testes p. 155

1 Introdução

O número de serviços em uma rede local cresce a cada dia. No mesmo ritmo, aumenta a exigência dos usuários por serviços sempre disponíveis, ou seja, acessíveis e funcionais. Os usuários finais estão tornando-se mais sensíveis e críticos quanto a indisponibilidade dos serviços que usam, mesmo que por pequenos intervalos de tempo. Portanto, as falhas nesses serviços, gerando mau funcionamento, problemas de conexão, dentre outros tipos de anomalias, devem ser o mais transparentes possíveis aos clientes. Neste sentido, um dos desafios das equipes de Tecnologia da Informação e Comunicação (TIC) é buscar a disponibilidade contínua dos serviços prestados no domínio local. Para atingir melhores níveis de disponibilidade, a complexidade de gerência dos serviços aumenta naturalmente. Cada vez mais os gerentes de redes precisam aumentar seu conhecimento e melhorar a organização das redes que gerenciam.

As falhas que ocorrem, tanto em nível de *hardware* como de *software*, são inevitáveis. É possível previní-las e diminuí-las, mas não eliminá-las por completo. Portanto, é preciso prover maneiras de contornar de forma eficiente e eficaz a ocorrência de falhas. Como meio para aumentar a disponibilidade dos sistemas, é comum serem aplicadas técnicas de redundância, tanto em *hardware* como em *software*. Em ambientes que possuem redundância em nível de serviços, é possível utilizar as réplicas em caso de falha das instâncias primárias. Dessa forma, os serviços tornam-se mais robustos e tolerantes a falhas.

A implementação prática de redundância por si só não garante a alta disponibilidade dos serviços. Para exemplificar, podem ser tomados como exemplo os serviços de DNS e LDAP, ambos com replicação nativamente implementada. Logo, é relativamente comum administradores de redes configurarem réplicas desses serviços, aumentando a tolerância a falhas.

O DNS é um serviço cujo servidor primário e secundário são geralmente configurados nos próprios sistemas clientes, presentes nos computadores e dispositivos dos usuários. Em casos de falha do servidor primário, a detecção, por parte dos clientes, costuma demorar vários segundos, ou até minutos, o que gera desconforto e insatisfação para os usuários.

No caso do LDAP, o problema é um pouco diferente do DNS. As aplicações que utilizam os serviços de diretório dos servidores LDAP normalmente são configuradas estaticamente e apontam para uma das réplicas do serviço. Com isso, se a réplica apresentar falhas ou ficar indisponível, as aplicações ficarão sem ter acesso aos dados fornecidos pelo LDAP. Isso, consequentemente, pode levar a indisponibilidade dos sistemas que dependem do LDAP para executar as suas funções. Esses problemas são realidade em muitas organizações, como é o caso do Núcleo de Tecnologia da Informação e Comunicação (NTIC) da Universidade Federal do

Pampa (UNIPAMPA).

Dado o contexto apresentado, a motivação para o trabalho é a falta de soluções simples e práticas para aumentar a disponibilidade de serviços em ambientes de redes locais, comumentemente encontrados em empresas públicas e privadas. Nesse sentido, o objetivo do trabalho é analisar e projetar uma solução para melhorar a disponibilidade dos serviços de rede com redundância implementada, melhorando de forma automatizada e transparente o acesso às réplicas dos serviços por parte dos clientes (sejam pessoas ou sistemas). Objetiva-se utilizar um cenário similar ao da UNIPAMPA como caso de estudo prático da solução proposta e implementada neste trabalho.

A solução que foi arquitetada e proposta é um projeto de *software* livre que destina-se a melhorar os índices de disponibilidade em redes locais que utilizam *firewalls* GNU/Linux, responsáveis por realizar a conexão entre os serviços e seus clientes, o que é uma configuração típica e presente em várias organizações, como ocorre na UNIPAMPA. A idéia central da solução é monitorar a disponibilidade das réplicas e ativar regras no *firewall* que façam o re-roteamento de pacotes, destinados a servidores indisponíveis, para réplicas operacionais e capazes de atender as requisições dos clientes.

Com uma arquitetura dinâmica, baseada na utilização de *plugins* e ferramentas externas para a realização de diferentes tarefas, como o monitoramento de serviços, manipulação de regras no *firewall*, envio de notificações e *log* de informações, a solução mostra-se extensível e adaptável às necessidades de diferentes cenários de redes locais que contenham serviços com replicação implementada.

No Capítulo 2 é apresentado o estudo realizado acerca de algumas soluções disponíveis e utilizadas, em cenários específicos, para o aumento da disponibilidade de serviços. Este estudo ajudou a identificar pontos fortes e fracos das soluções, o que possibilitou delinear as características pretendidas para a solução proposta. Os detalhes desta, incluindo seus requisitos, características, arquitetura e módulos que a compõem, estão descritos no Capítulo 3. O Capítulo 4 trata dos aspectos de implementação da solução proposta, incluindo as tecnologias utilizadas. O Capítulo 5 detalha os testes que foram realizados e apresenta os resultados obtidos. Por fim, as considerações finais e os trabalhos futuros são discriminados no Capítulo 6.

2 Revisão Bibliográfica

A alta disponibilidade dos serviços prestados é de crescente importância. Os usuários toleram cada vez menos as falhas dos sistemas que utilizam. Sendo assim, surgiram algumas soluções para garantir níveis mais altos de disponibilidade dos serviços prestados. Estas soluções convergem para a aplicação, de forma automática, das seguintes técnicas:

1. Redundância;
2. Balanceamento de carga;
3. Monitoramento, descoberta e tratamento de falhas.

Algumas das principais soluções encontradas que visam o aumento da disponibilidade de serviços, ambas livres e a nível de *software*, serão descritas neste capítulo:

1. Linux-HA (`http://www.linux-ha.org/`);
2. Linux Virtual Server (LVS) (`http://www.linuxvirtualserver.org/`);
3. Ultra Monkey (`http://www.ultramonkey.org/`);
4. Nagios (`http://www.nagios.org/`);
5. Zabbix (`http://www.zabbix.com/`).

Estas soluções são, na verdade, complementares e compartilham, por vezes, das mesmas ferramentas de *software*. Foram selecionadas apenas soluções que executam em ambientes GNU/Linux, pois atualmente é um dos sistemas mais utilizados para servidores e, portanto, mais relevante para este tipo de aplicações.

2.1 Linux-HA

Linux-HA é um acrônimo para “Linux High Availability”, que traduzido do inglês ficaria algo como “linux com alta disponibilidade”. Este é um projeto que trabalha em cima de uma solução para prover alta disponibilidade em ambientes GNU/Linux através da utilização de *clusters*[1]. Linux-HA permite explorar o potencial da redundância de serviços, porém, não

[1] Conjunto de computadores independentes que trabalham juntos em um propósito e são vistos em muitos aspectos como um único sistema. (WIKIPEDIA-CLUSTER, 2011)

provê redundância para aqueles que não a possuem nativamente implementada. Para os casos de serviços sem redundância nativa, devem ser utilizadas ferramentas alternativas, como o Distributed Replicated Block Device (DRBD) (`http://www.drbd.org/`), que faz o espelhamento de um dispositivo de bloco completo (DRDB, 2011). Em suma, para cada serviço deverá ser considerada uma alternativa de solução complementar e específica, aumentando a complexidade e impactando na administração dos sistemas. Linux-HA está em sua versão 2. Sendo assim, este documento fará referências a características e funcionalidades desta versão da solução.

Como um gerenciador de cluster o Linux-HA monitora a disponibilildade de seus membros (nós) e de seus serviços, podendo tratar situações de falha de forma autônoma e configurável. Há suporte para configurações ativo/ativo[2] e ativo/passivo[3]. No caso da configuração ativo/passivo, quando o servidor falho tornar-se novamente disponível, é possível realizar *failback*, ou seja, tornar o servidor novamente ativo, ou pode-se simplesmente mantê-lo como servidor passivo.

Uma funcionalidade importante que o Linux-HA não possui é o balanceamento de carga. Porém, esta funcionalidade pode ser provida por outras ferramentas, como por exemplo o HAProxy (`http://haproxy.1wt.eu/`) e o LVS. O HAProxy, no entanto, é dedicado para aplicações HTTP, embora seja capaz de balancear carga para outros serviços TCP (HAPROXY, 2011).

Um aspecto negativo do Linux-HA é a necessidade de ter um *cluster* próprio, o que faz necessário sua instalação e configuração em todos os nós do *cluster*. Detalhes sobre a arquitetura do Linux-HA encontram-se no Apêndice A. Conforme descrito no Apêndice B, a configuração do Linux-HA consiste na configuração individual de seus componentes, notadamente o Heartbeat (`http://linux-ha.org/wiki/Heartbeat`), responsável pela camada de comunicação, e o Pacemaker (`http://www.clusterlabs.org/wiki/Pacemaker`), gerenciador dos recursos do cluster. Isto gera redundância de informações de configuração como, por exemplo, a declaração dos nós do *cluster*. Desta forma, há a necessidade de que estas informações estejam consonantes em todos os membros.

2.2 Linux Virtual Server (LVS)

O LVS é uma solução para ambientes GNU/Linux que permite que um conjunto de máquinas que prestam serviços (*cluster*) seja visto como um único servidor, com um único endereço IP, a fim de prover altos níveis de disponibilidade e escalabilidade destes serviços. Seu nome

[2]Configuração na qual mais de uma réplica atende requisições em um determinado momento.
[3]Configuração na qual somente uma réplica atende requisições em um determinado momento.

traduzido do inglês, fica "Servidor Virtual Linux". Ele cria um servidor virtual, que serve de interface para todos os servidores reais do *cluster* (LVS, 2011), pois é através dele que os clientes acessam, de forma transparente, os serviços. O LVS funciona como um comutador lógico nível 4, pois recebe conexões, e repassa para seus destinos, registrando os endereços de origem e destino em uma tabela que permite identificar qual servidor está atendendo qual cliente.

O principal componente do LVS é o balanceador de carga, um software chamado IP Virtual Server (IPVS). O IPVS executa dentro do *kernel* (núcleo) do sitema operacional GNU/Linux, o que garante maior eficiência, ou então pode ser executado como um módulo carregável. As versões do *kernel* atualmente suportadas pelo IPVS vão desde a 2.2 até 2.6. Para configurar o IPVS existe uma ferramenta chamada IPVSADM. Esta é uma ferramenta em linha de comando que adiciona, remove e configura os servidores e serviços a serem atendidos pelo LVS.

A arquitetura de um sistema LVS possui duas camadas:

1. **Diretor**: é a interface para os serviços disponibilizados na rede. Para acessar um serviço, um cliente obrigatoriamente terá que acessar o diretor, que é a máquina responsável pelo balanceamento de carga. Basicamente, o diretor recebe as requisições dos clientes, define um servidor para atendê-lo através de um algoritmo de escalonamento e redireciona a conexão para ele. O balanceador de carga mantém uma tabela de escalonamentos, que é utilizada para determinar o servidor que atenderá cada requisição. É graças ao diretor que consegue-se fazer com que o *cluster* de servidores seja visto pelos clientes como um único servidor, de forma totalmente transparente.

2. **Parque de servidores**: é um *cluster* de servidores que fornecem os serviços de rede.

Graças a esta arquitetura, na qual os clientes enxergam um único endereço IP para todos os serviços, fica bastante fácil atingir altos níveis de escalabilidade e disponibilidade, já que as operações realizadas, como adição e remoção de nós, são transparentes aos clientes e aos próprios membros do *cluster*, sendo o único ponto de gerência o próprio diretor.

A escalabilidade é facilmente atingida por adicionar ou remover nós de acordo com a necessidade. Quando a carga saturar a capacidade do *cluster*, basta adicionar um novo nó, assim como quando o *cluster* estiver subutilizado, alguns servidores podem ser removidos, evitando que o balanceador de carga venha a tornar-se um gargalo no sistema por ter que manipular um número grande de nós. No entanto, esta tarefa fica a cargo do administrador, ou de alguma outra ferramenta capaz de monitorar as informações de carga fornecidas pelo LVS e inserir ou remover nós, conforme o que for adequado.

O LVS não faz verificação da situação dos servidores, portanto, ele não sabe nada a respeito

de eventuais falhas que venham a ocorrer. Sendo assim, para atingir alta disponibilidade é necessário utilizar uma ferramenta que monitore os servidores e serviços e reconfigure o LVS em caso de falhas, impedindo que novas conexões sejam criadas para os nós problemáticos. Com este objetivo, mais de uma ferramenta foi criada.

Um exemplo é o Ldirectord, um daemon escrito em Perl que faz checagens periódicas dos serviços e utiliza o IPVSADM para reconfigurar o LVS (HORMAN, 2011). O Ldirectord possui uma configuração bastante simples, com poucos parâmetros de configuração. Outra ferramenta que pode ser utilizada é o Keepalived, um *daemon* escrito em C (linguagem de programação) que monitora os serviços e o próprio diretor, provendo uma maneira de direcionar o diretor para um *backup* em caso de falha. O Keepalived utiliza *setsockopt* (KEEPALIVED, 2011), uma maneira de definir valores de opções para *sockets*, para reconfigurar o LVS. Sua configuração é um pouco complexa, com bastante opções de configuração.

Uma vantagem do LVS é possuir um grande número de algoritmos de escalonamento, e a possibilidade de utilizar diferentes algoritmos para diferentes servidores, o que possibilita escolher o mais adequado para cada serviço. A relação dos algoritmos de escalonamentos suportados encontra-se no Apêndice C.

Com o LVS é possível utilizar métodos diferentes de roteamento, incluindo tunelamento IP, que lhe atribui a possibilidade de gerenciar servidores distribuídos geograficamente (fora da rede local). Os métodos de roteamento suportados estão especificados no Apêndice D.

Um aspecto negativo do LVS é sua dificuldade de instalação e configuração, que exigem vários conhecimentos pertinentes, principalmente a ambientes GNU/Linux e redes TCP/IP. Sua grande dependência da versão do *kernel* também consiste em um empecilho, pois dificulta a atualização de ambos, *kernel* e LVS.

2.3 Ultra Monkey

Ultra Monkey é uma solução também destinada a ambientes GNU/Linux. Este, na verdade, é um pacote de componentes para prover alta disponibilidade e escalabilidade em *clusters*. O Ultra Monkey é composto pelas duas soluções anteriormente apresentadas, a Linux-HA e a LVS, utilizando também o Ldirectord. Sendo assim, ela representa uma das soluções mais completas para alta disponibilidade em *clusters*.

Seu objetivo é conciliar Linux-HA com LVS para obter uma solução a um problema que pode ocorrer no LVS: o balanceador de carga (diretor) tornar-se um ponto único de falha (ULTRAMONKEY, 2011). Para evitar este problema, o balanceador é replicado em um servidor

de *backup* e, através de uma configuração ativo/passivo, utiliza-se o Linux-HA para monitorar o serviço (ver Apêndice E). Quando o Heartbeat perder a comunicação com o servidor ativo, o servidor de *backup* deverá assumir seu lugar. Para isso, pode ser utilizado um endereço IP virtual flutuante (endereço que troca de máquina), o qual estará configurado sempre na interface de rede do servidor que estiver ativo. Então, quando o servidor primário falhar, este endereço virtual pode ser atribuído à interface de rede do servidor de *backup* e, através de ARP *spoofing*[4], os demais dispositivos da rede poderão atualizar o novo endereço físico correspondente ao endereço virtual. Após o ARP *spoofing*, todos os pacotes serão naturalmente direcionados para o servidor de *backup*.

Quando o servidor que falhou se recuperar, há duas alternativas possíveis: pode assumir o papel de servidor de *backup* ou pode voltar a ser o servidor primário (*failback*).

2.4 Nagios

O Nagios é uma das mais difundidas ferramentas para monitoramento de serviços e de máquinas. Seu maior propósito é informar problemas aos administradores de rede antes que os mesmos aconteçam, isto é, realizar previsões de problemas eminentes (NAGIOS1, 2011). Esta solução conta com uma variedade de funcionalidades, inclusive ferramentas capazes de realizar análise estatística e emitir relatório do nível de disponibilidade dos serviços.

Esta também é uma solução livre e foi desenvolvida para funcionar em ambientes Unix, o que engloba sistemas GNU/Linux. Sua instalação, inclusive, pode ser realizada através dos repositórios de algumas distribuições Linux. A instalação pelo código fonte é simples. Basta executar um *script* de configuração e depois o *makefile* (arquivo com instruções de compilação e instalação).

O Nagios opera como um centralizador de informações, pois seu núcleo é instalado e configurado em uma máquina da rede, que a partir de então, passa a realizar o monitoramento e coleta de informações das outras máquinas e serviços. Seu alvo principal são redes com grandes números de dispositivos a serem monitorados, porém, pode ser aplicado a pequenas redes.

Sua capacidade vai além de monitorar serviços de rede. Ele também é capaz de coletar informações como carga de processamento, utilização de memória, espaço em disco, temperatura, dentre outras, de máquinas remotas (NAGIOS1, 2011). Para tal, é necessário instalar agentes específicos nas máquinas onde deseja-se monitorar estas informações (HEIN, 2007).

[4]Técnica na qual um dispositivo informa seu endereço IP de forma voluntária, através de respostas ARP, fazendo com que as outras máquinas vinculem o endereço IP ao seu endereço físico.

Estes agentes, em grande parte, possuem versões para outros tipos de sistemas operacionais, como o Windows, por exemplo, o que amplia seu poder de monitoramento. Como o princípio norteador da solução é prever problemas antes que eles aconteçam, estas informações a respeito da utilização de recursos e situação de componentes físicos, consistem em alguns dos principais indicadores que servem de arcabouço para realização das previsões.

Para transmissão de dados remotos para o Nagios pode ser utilizado um *plugin* chamado *check_by_ssh*, que abre uma conexão SSH, executa o agente na máquina remota, obtém a resposta e transmite para o Nagios (HEIN, 2007). Outra alternativa é utilizar o *Nagios Remote Plugin Executor* (NRPE), uma aplicação cliente/servidor capaz de executar *plugins* remotamente e receber a resposta por uma conexão TCP estabelecida. O NRPE consiste em um servidor, que executa como um *daemon* nas máquinas a serem monitoradas, escutando na porta 5666, e um *plugin* cliente utilizado pelo Nagios (NAGIOS2, 2011). Uma vantagem de utilizar o NRPE é a possibilidade de monitorar serviços de rede em máquinas que não são atingíveis a partir do servidor onde encontra-se o Nagios, mas são atingíveis através da máquina sendo monitorada. Por fim, há a possibilidade de utilizar o *Simple Network Management Protocol* (SNMP), um protocolo de gerência de redes com a finalidade de intercâmbio de informações. Para utilização deste protocolo, há vários *plugins* disponíveis.

Não basta apenas identificar problemas eminentes, é necessário informá-los aos administradores. Sendo assim, o Nagios possui um rebuscado sistema de alertas que permite granularizar a emissão de avisos. Em sua configuração, é possível definir os grupos de contatos, os períodos, as regras e condições que devem ser satisfeitas para envio de alertas. Desta maneira, é possível evitar o envio de mensagens desnecessárias. Para a emissão dos alertas propriamente ditos, o Nagios utiliza *plugins*, o que confere maior flexibilidade, possibilitando que o próprio usuário desenvolva seus programas ou *scripts* de notificação.

Os verificadores de serviços também são *plugins*, e não fazem parte do núcleo do Nagios (SCHILLI, 2007). Para desenvolver verificadores de serviço, basta seguir a documentação, que contém os detalhes e as interfaces de comunicação do Nagios com seus *plugins*. De forma resumida, o Nagios avalia dois aspectos dos *plugins*. O primeiro, são mensagens formatadas enviadas para a saída padrão do programa. Estas mensagens contém os dados coletados durante a verificação e possíveis informações sobre erros ocorridos. O segundo aspecto utilizada pelo Nagios, é o código de saída do *plugin*, o qual determina o resultado da verificação. Por exemplo, código 0 significa que não foram encontrados problemas, código 1 significa atenção, pois alguma coisa não está dentro do normal, e assim por diante.

Uma funcionalidade bastante importante do Nagios, consiste na possibilidade de configu-

rar e invocar manipuladores de eventos quando da modificação de estado de um serviço ou servidor. Uma das principais aplicações deste recurso consiste em, proativamente, tentar corrigir os problemas encontrados, através da execução de agentes. Através desta funcionalidade é possível, inclusive, mover a incubência de monitoramento e emissão de alertas entre réplicas do Nagios, mantendo a disponibilidade do serviço em caso de falha das réplicas primárias. Há vários ajustes possíveis quanto a utilização de manipuladores de eventos. Estes ajustes basicamente consistem nas condições necessárias para que os manipuladores de eventos sejam ativados.

Para visualizar a situação atual dos componentes monitorados, existe uma interface *web* que vem integrada ao Nagios. Para utilizá-la, é necessário ter instalado um servidor *web*. Nesta interface é possível verificar, de maneira simples, informações sobre a disponibilidade dos servidores, estatísticas, agendamento de verificações, dentre várias outra informações relevantes.

Os valores de configuração do Nagios ficam todos armazenados em arquivos. A sintaxe dos arquivos é bastante simples e intuitiva, porém, devido a carga de informações a ser fornecida, a configuração torna-se complexa e, por vezes, de difícil manutenção. Por exemplo, quanto ao monitoramento, há um elevado número de aspectos que devem, e uma quantidade muito maior de informações que podem ser especificadas. Isso acaba por exigir uma gama de conhecimentos específicos do real funcionamento da aplicação, o que torna a tarefa de configurar massiva e complexa.

Com o objetivo de auxiliar na configuração, existem vários projetos de desenvolvimento de interface gráfica como um *front-end* para a configuração. Segundo (HEIN, 2007), estes projetos obtiveram variados graus de sucesso e alguns foram descontinuados e não acompanharam a evolução da ferramenta.

2.5 Zabbix

O Zabbix é uma solução para monitoramento de serviços e dispositivos de rede. Esta ferramenta possui licença GPL e é semelhante ao Nagios. Segundo (BLACK, 2008), tem sido comumente considerada uma ferramenta superior as demais ferramentas livres para monitoramento, pois concentra funcionalidades de várias outras, eliminando a necessidade de utilização de outras soluções em paralelo.

O Zabbix trabaha com conceitos comuns ao Nagios, como a utilização de agentes para coletar informações da máquinas monitoradas, definições para emissão de alertas, manipuladores de eventos, utilização de interfaces *web* para configuração e visualização da situação rede, emissão

de relatórios estatísticos, dentre outros.

Um ponto forte do Zabbix é a utilização de bancos de dados para armazenamento das informações coletadas. Segundo (BLACK, 2008) há suporte para a utilização de vários sistemas gerenciadores de banco de dados (SGBD), sendo eles MySQL, Oracle e Postgres, ambos bastante conhecidos e utilizados.

2.6 Análise das soluções

Dentre as soluções apresentadas, Nagios e Zabbix consistem em ferramentas para monitoramento de serviços e sistemas. Através da coleta e análise de informações dos dispositivos e serviços de rede, permitem que os administradores encontrem os problemas ocorridos, assim como possibilitam a previsão de falhas eminentes, porém, o foco da utilização delas não recai sobre a execução de ações paliativas a falhas encontradas, e sim na exibição da situação atual da rede.

Estas duas ferramentas possuem uma grande variedade de funcionalidades e recursos de monitoramento, porém, não encontrou-se um caso em que fossem utilizadas para realizar o monitoramento para as demais soluções. Por exemplo, seria interessante que o LVS utilizasse os *plugins* do Nagios para monitorar os serviços, ou então, o Linux-HA utilizar os agentes do Zabbix para coletar as informações dos serviços e passá-las ao gerenciador de *cluster*.

As soluções LVS, Linux-HA e Ultra Monkey conseguem explorar a redundância existente em *clusters* a fim de aumentar significativamente o grau de disponibilidade dos serviços prestados. Com o objetivo de realizar uma análise e identificar os pontos positivos e negativos de cada uma destas três soluções, foram estabelecidas algumas métricas e itens. As características analisadas foram:

1. **Complexidade de instalação**: refere-se a quantidade de conhecimento e passos necessários para efetuar a instalação. Linux-HA possui pacotes pré-construídos para várias distribuições GNU/Linux. Através destes pacotes, a instalação torna-se simples, bastando utilizar um gerenciador de pacotes da distribuição. A instalação através do código fonte também é simples, pois possui *scripts* de configuração e *makefiles* que necessitam apenas ser executados. Sua desvantagem é a necessidade de ser instalado em todas as máquinas do *cluster*. O LVS e o Ultra Monkey, em contrapartida, exigem um bom conhecimento da arquitetura e funcionamento do sistema operacional GNU/Linux, visto que, para serem instalados, necessitam que o *kernel* seja recompilado, ou que um módulo externo seja construído e carregado. A vantagem do LVS e Ultra Monkey é necessitarem de instalação somente

nos diretores.

2. **Complexidade de configuração**: refere-se a quantidade de conhecimento e procedimentos necessários para configurar a solução e torná-la operacional. Linux-HA exige um grau bastante alto de detalhes sobre a configuração e funcionamento desejado. No caso do Linux-HA ainda há o fator agravante de a configuração de seus componentes ser separada e possuir redundâncias de informações, que, logicamente, devem ser consonantes. O LVS possui, nativamente, apenas uma ferramenta em linha de comando para configurar o diretor, porém, existem ferramentas externas que auxiliam nesta tarefa. De acordo com o modo de roteamento selecionado, pode ser necessária uma configuração de rede um pouco mais complexa nos servidores do *cluster*. O LVS possui também uma ferramenta que busca automatizar o processo de configuração através da geração de um *script* de configuração, porém, este *script* somente configura o balanceador de carga e ajusta a configuração de rede dos servidores do *cluster*, se utilizarem GNU/Linux. Para o monitoramento dos serviços são utilizadas ferramentas externas (LdirectorD, KeepaliveD), as quais são configuradas independentemente. O Ultra Monkey, por ser composto pelo LVS e o Linux-HA, possui seus mesmos prós e contras.

3. **Gerenciamento (localidade)**: está relacionado ao local da rede onde as tarefas de gerenciamento da solução e de seus componentes são realizadas. O LVS realiza o gerenciamento no diretor, de forma centralizada, enquanto que Linux-HA distribui esta tarefa entre os servidores do *cluster*. O Ultra Monkey possui mais de um diretor, logo, seu gerenciamento é distribuído.

4. **Sistemas operacionais nos servidores monitorados**: avalia os possíveis sistemas operacionais suportados pelos servidores que serão monitorados pela solução. O Linux-HA necessita ser instalado nos servidores que monitora, logo, somente são suportados sistemas GNU/Linux. No caso do LVS e do Ultra Monkey, os sistemas operacionais somente necessitam suporte à pilha de protocolos TCP/IP, pois são utilizadas as interfaces dos serviços para o monitoramento. De acordo com o modo de roteamento utilizado, outros fatores podem influenciar.

5. **Balanceamento de carga**: neste item as soluções são avaliadas quanto a capacidade de fazer balanceamento de carga entre os servidores monitorados. Somente o Linux-HA não possui esta habilidade.

6. **Focada em *clusters***: define se as soluções foram projetadas com enfoque na utilização em *clusters*. Linux-HA e Ultra Monkey somente funcionam em *clusters*, tanto que possuem

cluster específico. O LVS foi projetado para atender *clusters*, embora possa ser utilizado em redes convencionais.

7. **Intrusividade**: avalia o quão intrusiva é a adoção da solução, isto é, o impacto e alterações necessárias na configuração da rede e dos servidores para comportar a solução. O Linux-HA necessita de modificação nos servidores para que seja criado seu próprio *cluster*. Quanto ao LVS, a intrusividade varia de acordo com o método de roteamento utilizado. O método menos intrusivo é o NAT, que necessita apenas a mudança do *gateway* padrão nos servidores, que deve passar a ser o próprio diretor. A própria adição do diretor altera a configuração da rede. O Ultra Monkey prevê ainda a adição de um diretor de *backup*.

Através da Tabela 2.1, onde estão sintetizadas as características que foram analisadas das soluções Linux-HA, LVS e Ultra Monkey, percebe-se que são destinadas a ambientes específicos (*clusters*). Outras características relevantes são a intrusividade elevada e a complexidade de instalação e ou configuração, o que pode gerar custos e diminui a aceitabilidade das soluções para ambientes convencionais de redes de pequeno e médio porte. Desta análise, conclui-se que as soluções Linux-HA, LVS e Ultra Monkey possuem algumas características que consistem em um entrave para implantação em ambientes de rede típicos, onde não há a existência de *clusters* e há menos sensibilidade a níveis menores de disponibilidade, o que pode acarretar de não ser justificável a implantação destas soluções.

Tabela 2.1 – Síntese das características analisadas no Linux-HA, LVS e Ultra Monkey

	Linux-HA	**LVS**	**Ultra Monkey**
Complexidade de instalação	Baixa	Alta	Alta
Complexidade de configuração	Alta	Média/Alta	Alta
Gerenciamento (localidade)	Distribuído	Centralizado	Distribuído
SOs nos servidores monitorados	GNU/Linux	Qualquer um*	Qualquer um*
Balanceamento de carga	Não	Sim	Sim
Focada em *clusters*	Sim	Sim	Sim
Intrusividade	Alta	Média/Alta	Média/Alta

Fonte: do autor

* Com suporte à pilha de protocolos TCP/IP. Depende do modo de roteamento utilizado também.

3 A Solução Proposta

As soluções de alta disponibilidade encontradas possuem características e peculiaridades que as tornam complexas ou inadequadas a ambientes convencionais de redes locais. Algumas delas são focadas para *cluster* de serviços, o que não é o caso de grande parte das redes locais. Outras implicam em alterações arquiteturais ou organizacionais na rede, ou exigem a instalação de componentes e alteração de sistemas. Como as redes locais já são complexas por si só, soluções mais complexas, soluções que implicam em alterações de arquitetura ou organização da rede, ou ainda soluções que implicam em conhecimentos avançados e grandes curvas de aprendizado são pouco desejáveis. Na prática, soluções como as citadas no Capítulo 2 são pouco difundidas e utilizadas em redes locais convencionais por esses motivos, entre outros.

Em redes de pequeno porte, é comum uma configuração onde é utilizado um ou mais *firewalls* que servem, dentre outras coisas, para rotear os pacotes entre os servidores da rede local e seus clientes externos. Por vezes, também são utilizados para realizar a comunicação entre os servidores e as demais máquinas da rede interna, principalmente por questões de segurança. Esta é uma configuração típica e utilizada em várias organizações.

No caso do NTIC da UNIPAMPA, é possível constatar o cenário anteriormente descrito. Sua rede local é constituída de um pequeno grupo de servidores, os quais são acessíveis externamente através de um *firewall*, responsável pela política de segurança. Alguns dos serviços disponibilizados estão replicados em um ou mais servidores, como é o caso do DNS, SMTP e LDAP. Os clientes (aplicações de usuários e outros sistemas) são configurados para acessar diferentes conjuntos de réplicas dos serviços. Isso contribui na distribuição de carga e, em tese, na disponibilidade dos serviços aos clientes. No entanto, como a configuração dos clientes é manual na maior parte dos casos, a indisponibilidade de algumas réplicas pode deixar alguns clientes sem o serviço. O resultado, naturalmente, é transformado em insatisfação, além de ser uma prática que dificulta ainda mais o gerenciamento dos sistemas e clientes devido a configuração manual.

Outro caso ainda mais grave é quando o cliente é configurado para ter acesso a uma única réplica, ou seja, ter várias réplicas não melhora em nada o índice de disponibilidades para esses clientes. E esse tipo de cliente é uma realidade comum em organizações de pequeno e médio porte. Nesse sentido, o objetivo da solução aqui proposta é justamente atacar e resolver esse problema, ou seja, evitar que os clientes fiquem sem acesso aos serviços replicados enquanto pelo menos uma das réplicas estiver ativa, funcionando corretamente. Esse acesso às réplicas ativas deve ser realizado de forma automática e transparente pelos clientes. Estes não devem

sequer perceber que estão utilizando outras réplicas.

Visando melhorar os níveis de disponibilidade de uma forma simples e fácil, em ambientes de rede onde há redundância de serviços e *firewalls*, foi proposto o desenvolvimento de uma nova solução. O nome escolhido para a solução é *AvailableD*. “Available” é uma palavra inglesa que siginifica “disponível” e a letra “D” do nome faz referência a palavra *daemon*, que é o nome dado a programas que executam por longos períodos de tempo em *background* (sem interação com o usuário), o que consiste em uma característica da ferramenta proposta.

3.1 Visão Geral

Configurações estáticas são muito comuns em aplicações que acessam os serviços. O problema fica ainda maior quando os clientes são outros sistemas, que possuem os endereços informados em arquivos de configuração e, por vezes, no próprio código fonte. Nestes casos, para permitir o acesso às réplicas ativas, seria necessário acessar os arquivos e realizar a troca manual dos endereços.

O serviço DNS é um exemplo clássico no qual os clientes, geralmente aplicativos para configuração de rede, são configurados com os endereços dos servidores primário e secundário. Embora a existência de um servidor alternativo seja prevista, há um considerável intervalo de tempo (conhecido como *timeout* do protocolo ou da implementação do aplicativo) para que o secundário seja consultado em caso de falha do primário. Isso deve-se ao fato que uma consulta DNS pode realmente ser um pouco demorada, pois o tempo de comunicação entre cliente e servidor, principalmente através de grandes redes, como a internet, pode ser prejudicada por vários fatores, como uma longa distância física. Dessa forma, pode ser necessário um tempo na ordem de alguns segundos, podendo chegar a um minuto ou mais, até a aplicação considerar o servidor primário como inoperante. Apenas após esta detecção o servidor secundário será consultado. Isto demonstra que não é o suficiente conhecer o servidor secundário, pois, mesmo assim, pode levar a insatisfação do usuário.

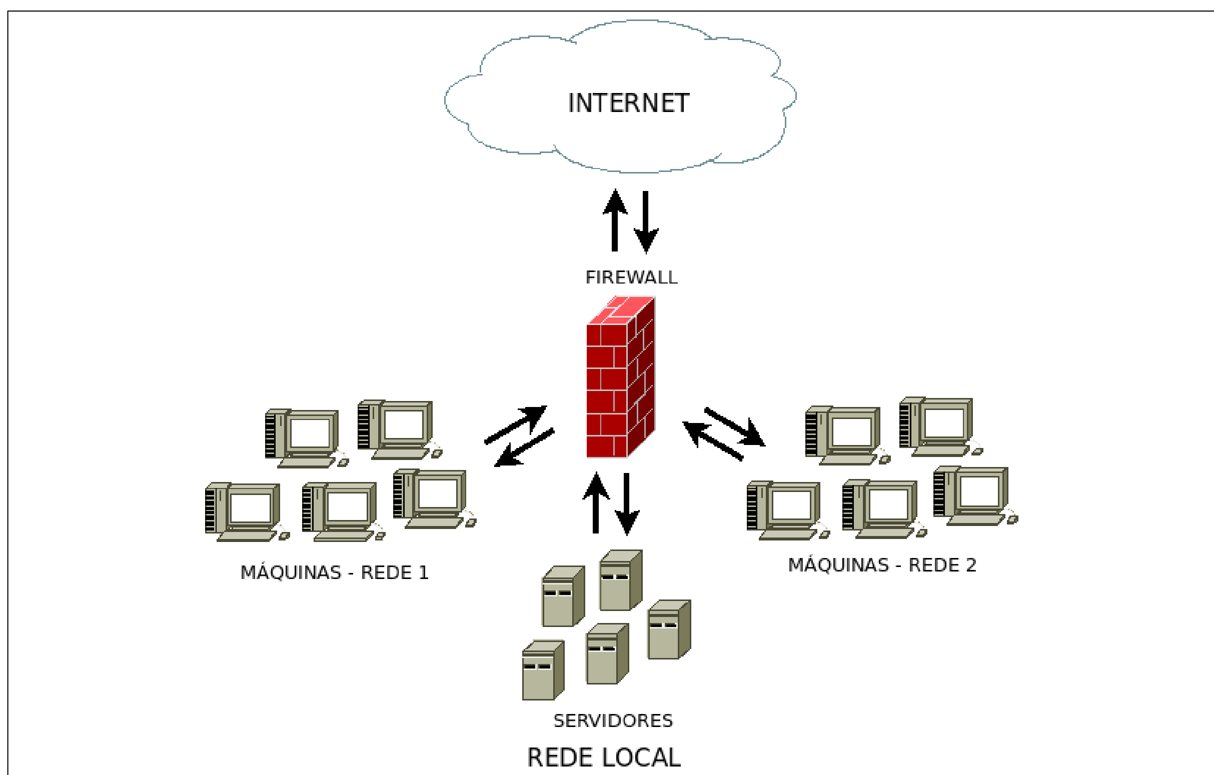
Considerando o problema das configurações estáticas no lado dos clientes, o principal artifício utilizado pelo AvaialableD para elevar o índice de disponibilidade é a automação da adição e remoção de regras de redirecionamento de pacotes em *firewalls*, de acordo com os dados provenientes do monitoramento da disponibiliadade dos serviços. Desta forma, as eventuais falhas em algumas instâncias dos serviços serão menos perceptíveis, elevando o nível de satisfação dos usuários com os serviços prestados de uma forma transparente.

A transparência aos clientes com relação a réplica que está prestando o serviço é um ponto

chave, pois possibilita que o gerenciamento da configuração de endereços de rede concentre-se no lado dos servidores. Isso significa que os clientes não precisam preocupar-se com os endereços das réplicas dos serviços. Para o cliente, é o suficiente ter o endereço de uma única réplica. Quando a réplica do respectivo endereço falhar, as requisições enviadas para ela serão redirecionadas automatica e transparentemente, no *firewall*, para o endereço de uma das réplicas ativas e funcionais. Essa abordagem também reduz os custos de manutenção de configurações no lado do cliente, pois o *AvailableD* irá tomar todas as medidas para garantir que cada cliente continue transparentemente acessando uma das réplicas do serviço.

A solução do *AvailableD* foi pensada e projetada para sistemas GNU/Linux. A justificativa reside na grande utilização destes sistemas como *firewalls* em redes locais. A figura 3.1 ilustra uma arquitetura de rede baseada em um *firewall* central, similar aos ambientes alvo da solução proposta. Nela, percebe-se que a ponte de comunição entre os clientes e os servidores da rede consiste no *firewall*.

Figura 3.1 – Ambiente alvo de aplicação da nova solução proposta



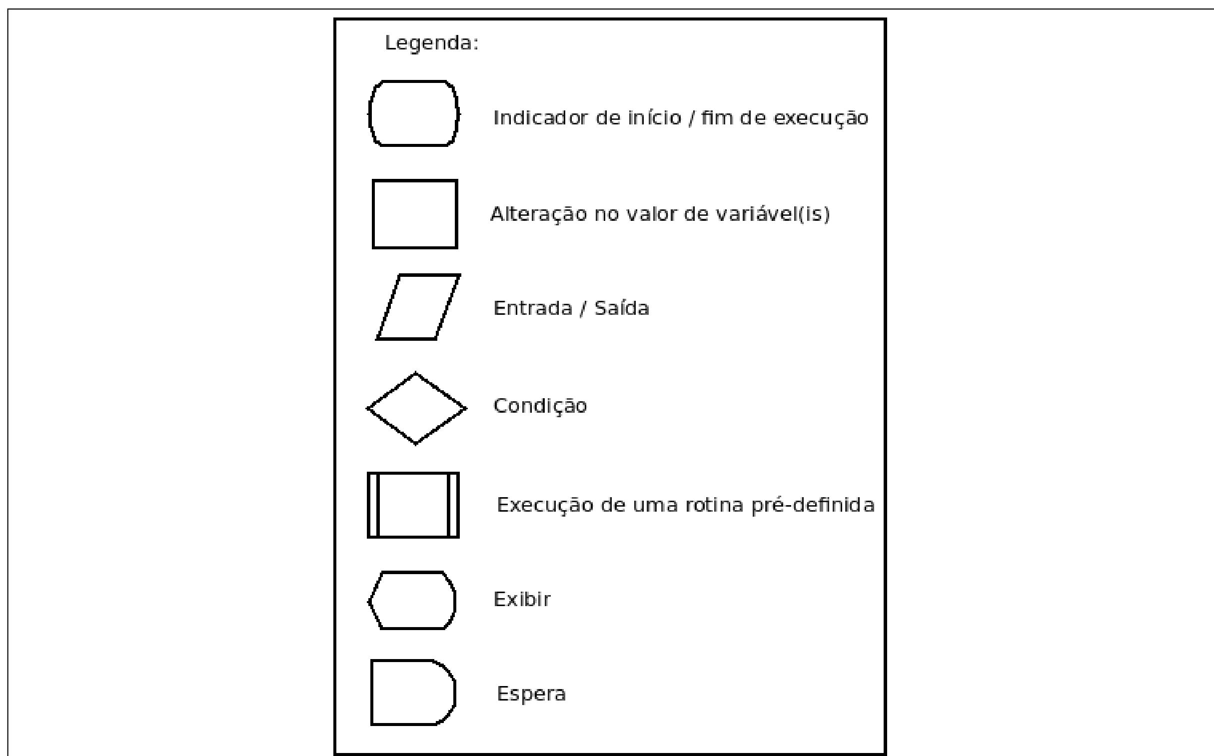
Fonte: do autor

Em linhas gerais, o funcionamento simplificado do *AvailableD* consiste em carregar sua configuração, iniciar o monitoramento dos serviços e, sempre que um serviço tornar-se indisponível, ou voltar ao estado de disponibilidade, são alteradas as regras de roteamento no *firewall*, os administradores da rede são notificados e é realizado *log*. O funcionamento está ilustrado no fluxograma da Figura 3.3. A legenda para as figuras deste documento que ilus-

tram funcionamento através de fluxogramas de execução, encontra-se na Figura 3.2. As Figuras 3.4, 3.5 e 3.6 mostram um exemplo prático do funcionamento do *AvailableD*. Nestas figuras, o serviço sendo monitorado é o HTTP, redundante, ou seja, dois servidores *web* (IP 10.1.1.1 e porta 80 e IP 10.1.1.1 e porta 8080) executando os mesmos serviços. É ilustrado o funcionamento da solução nas seguintes situações: os dois servidores estão ativos (Figura 3.4), detecção de falha em um dos servidores (Figura 3.5) e o momento quando ele volta a estar disponível (Figura 3.6).

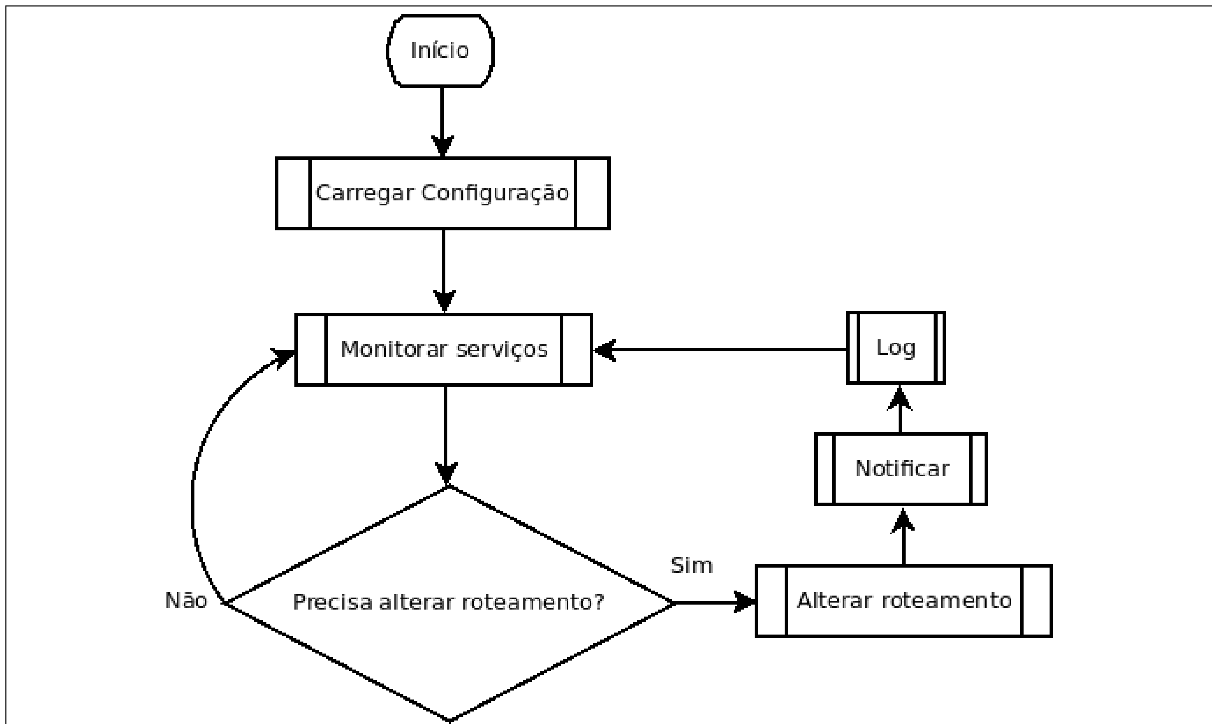
Nas Figuras 3.7 e 3.8 é possível visualizar o funcionamento pela perspectiva do cliente, quando requisições são feitas a servidores disponíveis e a servidores indisponíveis. Na Figura 3.7, o cliente faz uma solicitação HTTP ao servidor 10.1.1.1 na porta 80, que encontra-se funcional. Na Figura 3.8, após detectada uma falha na réplica e incluído um redirecionamento do IP 10.1.1.1, na porta 80, para o IP 10.1.1.2, na porta 8080, o cliente passa a acessar o segundo servidor HTTP de forma transparente. O cliente sequer fica sabendo que houve uma falha com o serviço prestado pela réplica no IP 10.1.1.1.

Figura 3.2 – Legenda para fluxogramas de execução



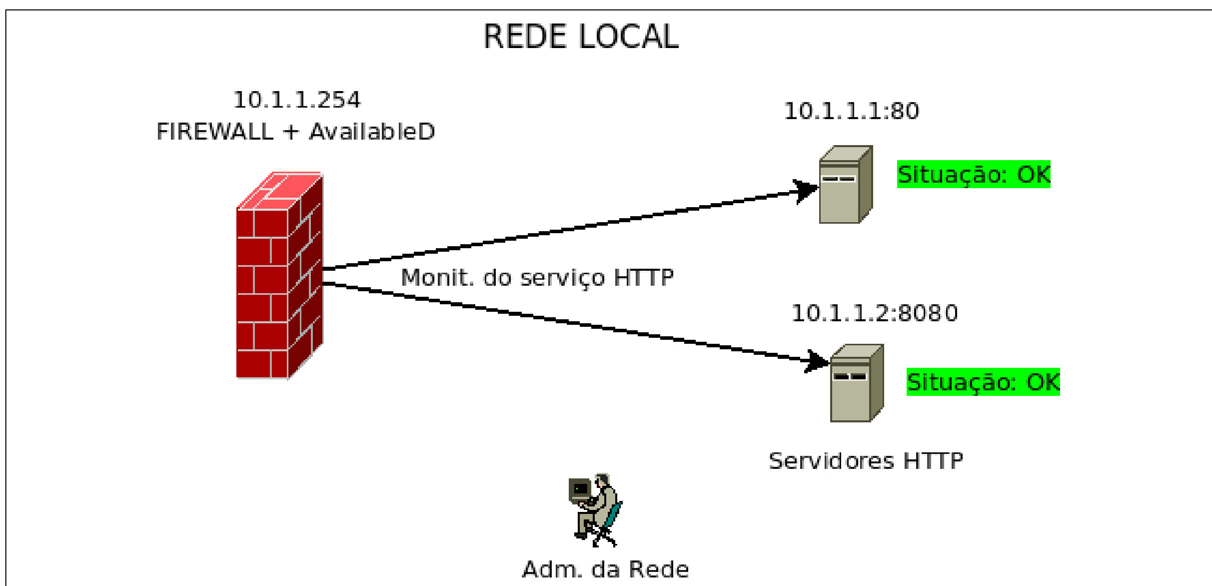
Fonte: do autor

Figura 3.3 – Funcionamento simplificado da solução *AvailableD*



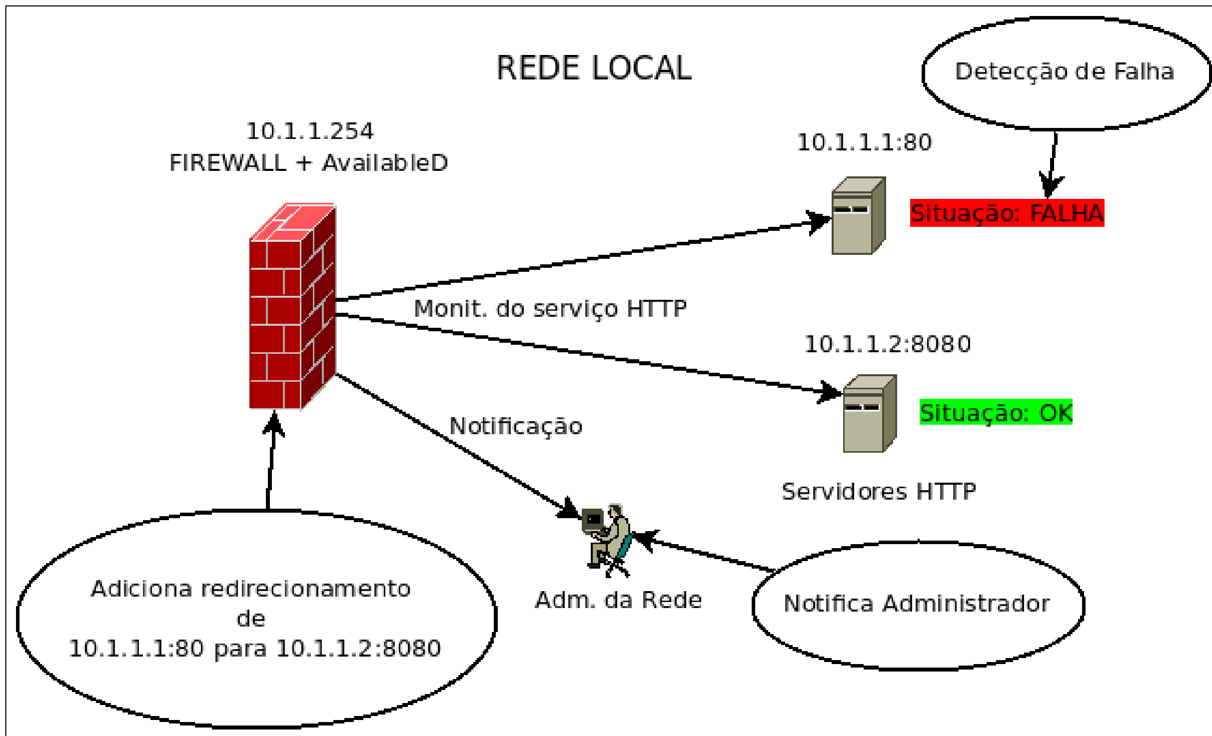
Fonte: do autor

Figura 3.4 – *AvailableD* monitorando o serviço HTTP em dois servidores



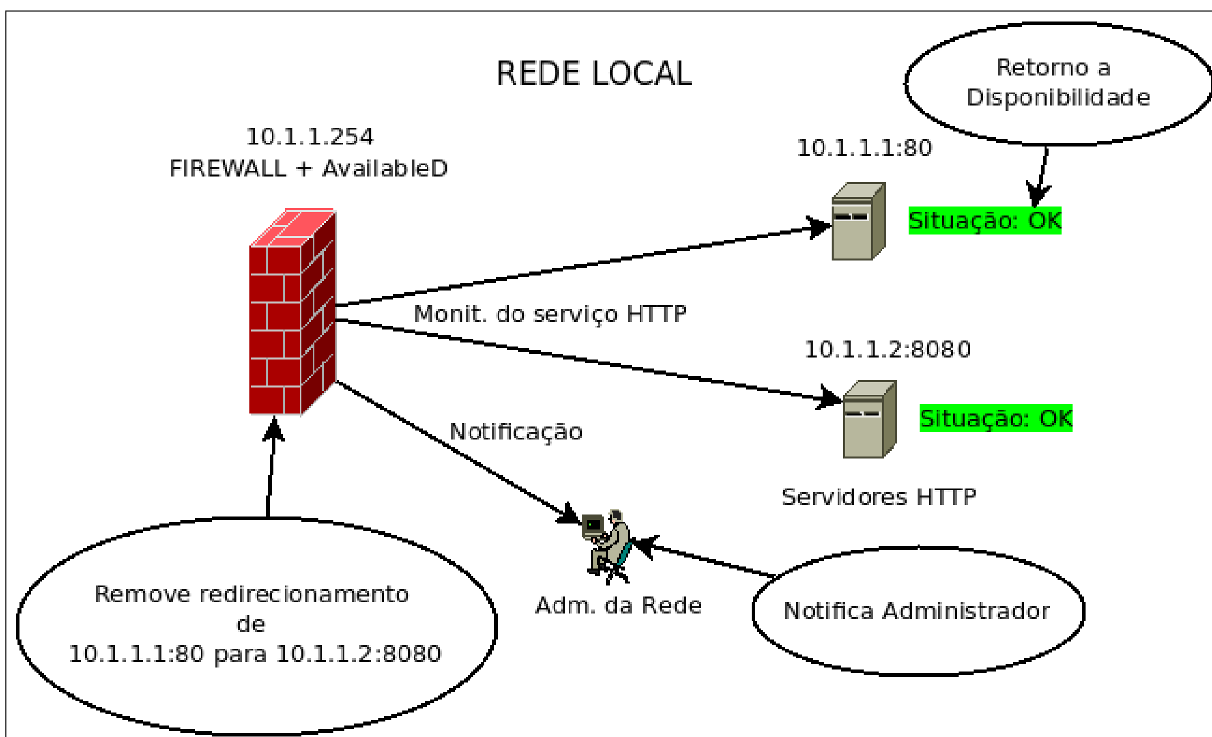
Fonte: do autor

Figura 3.5 – Redirecionamento realizado pelo *AvailableD* após identificação de falha



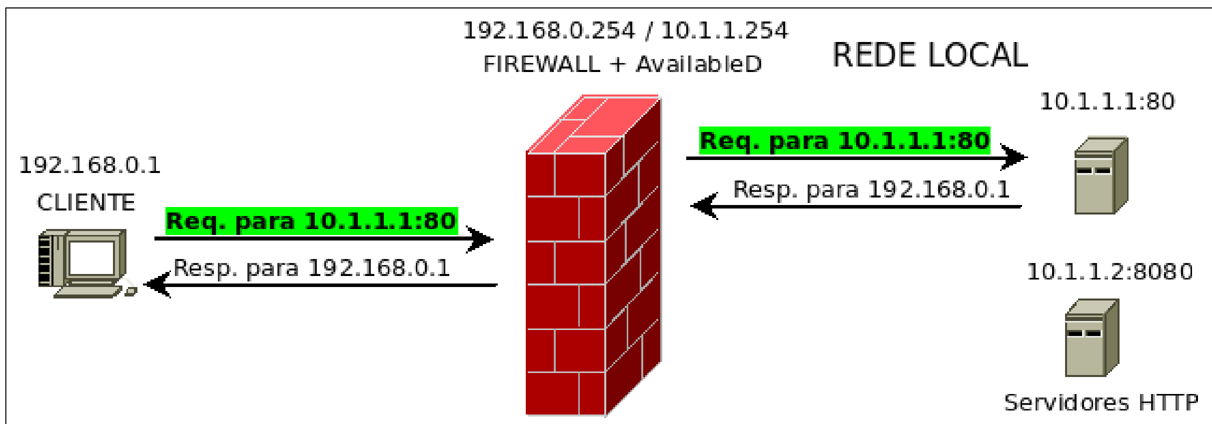
Fonte: do autor

Figura 3.6 – *AvailableD* removendo redirecionamento após o retorno a disponibilidade



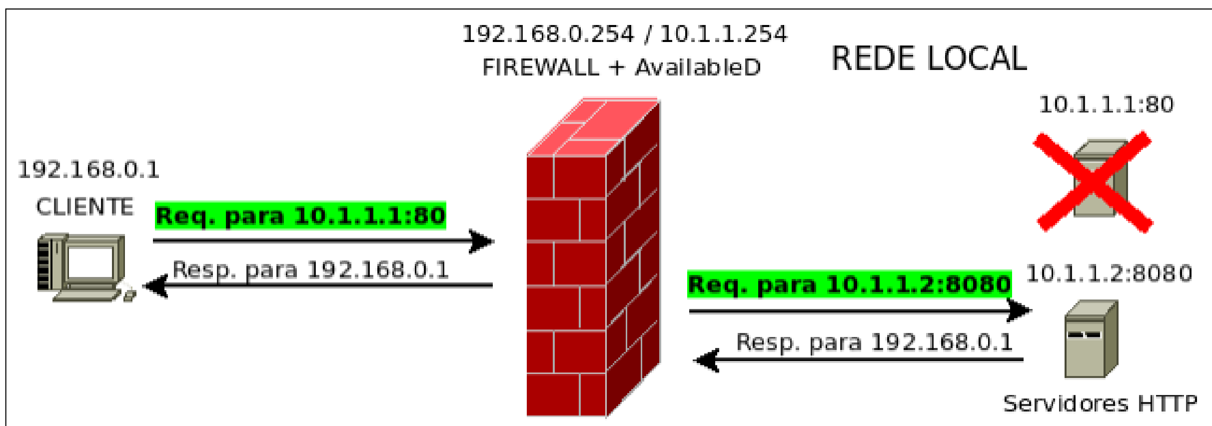
Fonte: do autor

Figura 3.7 – Requisição HTTP para um servidor disponível



Fonte: do autor

Figura 3.8 – Requisição HTTP para um servidor indisponível



Fonte: do autor

3.2 Premissas e Requisitos

Um dos princípios fundamentais de concepção da solução proposta é a simplicidade e facilidade de utilização por parte dos usuários alvo, ou seja, administradores de sistemas e redes. Com este objetivo, foram identificadas e definidas algumas premissas a serem levadas em consideração no projeto e na implementação da solução, como:

1. **Intelegibilidade**: a intenção é simplificar os conceitos utilizados pelo sistema, proporcionando o fácil entendimento dos mesmos por parte do usuário;

2. **Apreensibilidade**: através da utilização de padrões com os quais administradores de redes já estão acostumados, busca-se atingir uma curva de aprendizagem pequena no que tange ao uso da ferramenta;

3. **Operacionalidade**: para facilitar a manipulação do sistema, pretende-se simplificar e manter um conjunto reduzido de operações disponíveis;

4. **Facilidade de Instalação**: o objetivo é possibilitar que a solução seja instalada em poucos passos, fáceis de serem executados e bem definidos;

5. **Facilidade de Configuração**: a configuração deve ser simples e sucinta, possibilitando que em pouco tempo o usuário consiga configurar o sistema de acordo com suas necessidades;

6. **Baixa Intrusividade**: para que seja fácil sua implantação, a solução tem que se adaptar ao cenário ao qual é proposta. Logo, é fundamental que não seja necessário realizar alterações na estrutura nem na configuração da rede onde for utilizada.

O levantamento de requisitos, levando em consideração as principais premissas elencadas, resultou na identificação de cinco funcionalidades principais:

1. **Monitoramento**: realizar a verificação da disponibilidade dos serviços de rede;

2. **Regras de redirecionamento**: adicionar regras de redirecionamento das requisições destinadas a servidores indisponíveis para réplicas do mesmo serviço que estejam funcionais. Remover as regras de redirecionamento quando os servidores voltarem a estar disponíveis;

3. **Notificação**: noficar os administradores da rede sobre a mudança no estado dos servidores (disponível/indisponível);

4. **Log**: salvar dados das ocorrências de mudança no estado dos servidores, possibilitando uma posterior análise dos eventos ocorridos;

5. **Configuração**: possibilitar determinar os serviços a serem monitorados, os alertas a serem emitidos e os comandos de redirecionamento a serem executados.

Alguns requisitos não funcionais também foram identificados:

1. **Usabilidade**: a manipulação da ferramenta deve ser fácil;

2. **Simplicidade**: os conceitos envolvidos devem manter-se simples, de fácil entendimento;

3. **Desempenho**: o desempenho da solução deve atingir níveis aceitáveis, que não comprometam a utilização da máquina onde estiver sendo executada;

4. **Portabilidade**: desejável, porém o foco mantém-se em sistemas GNU/Linux;

5. **Confiabilidade**: a solução deve ser confiável o suficiente a ponto de eventuais erros não resultarem em consumo excessivo de recursos da máquina, geração de grande tráfego na rede ou até mesmo expor a máquina a problemas de segurança, principalmente por tratar-se de um *firewall*;

6. **Interoperabilidade**: deve ser possível monitorar serviços prestados em máquinas com sistemas operacionais diferentes, desde que seja possível utilizar a interface do serviço como meio de comunicação.

Os requisitos funcionais e não funcionais apresentados estão descritos em maiores detalhes no documento de requisitos, Apêndice F.

3.3 Arquitetura do AvailableD

A arquitetura geral do *AvailableD* é composta por seis grandes módulos, com diferentes finalidades.

1. **Configuração**: responsável por obter as informações de configuração do sistema e transmití-las aos demais módulos;

2. **Monitoramento**: realiza as verificações de disponibilidade dos serviços. Os verificadores utilizados pelo módulo são programas externos, ou seja, são *plugins* que são executados pelo módulo para obter a situação dos serviços, de forma análoga ao Nagios. O modelo baseado em *plugins* confere extensibilidade, além de permitir uma fácil adaptação de *plugins* de outras ferramentas, ou até mesmo o desenvolvimento dos próprios *plugins* por parte dos usuários. Outro benefício deste modelo é que consiste basicamente de executar programas e passar argumentos para eles, tarefa bastante simples que praticamente todo o administrador de redes está habituado a fazer;

3. **Executor de ações**: ativa e remove redirecionamentos, de acordo com a análise dos resultados das verificações realizadas pelo módulo monitoramento. As ações são configuráveis pelos usuário e consistem em programas externos, de modo semelhante ao módulo monitoramento;

4. **Log**: transmite mensagens recebidas pelos demais módulos a gerenciadores de *log*, responsáveis por armazenar estas informações. O objetivo é guardar o histórico da execução do sistema para ser analisado sempre que necessário;

5. **Notificação**: este módulo é responsável por emitir notificações aos administradores da rede alertando sobre falhas detectadas nos servidores, a fim de minimizar o tempo para correção dos problemas. Este módulo comporta-se de maneira semelhante aos módulos de monitoramento e executor de ações, pois também prevê a execução de programas externos (*plugins*) para a emissão das notificações.

6. **Integrador**: realiza a integração entre os demais módulos e é responsável pela lógica de operação do sistema.

A Figura 3.9 ilustra a arquitetura interna do *AvailableD* e sua interação com *plugins* e programas externos. Nela podem ser vistos os seis módulos e a comunicação entre eles. Os módulos configuração, monitoramento, executor de ações, log e notificação transmitem e ou recebem informações do módulo integrador, que é responsável por proceder a troca de informações entre eles. O módulo de log utiliza gerenciadores de *log* para armazenar as informações, ao passo que os módulos monitoramento, notificação e executor de ações utilizam *plugins* para atingir seus objetivos.

3.3.1 Módulo Configuração

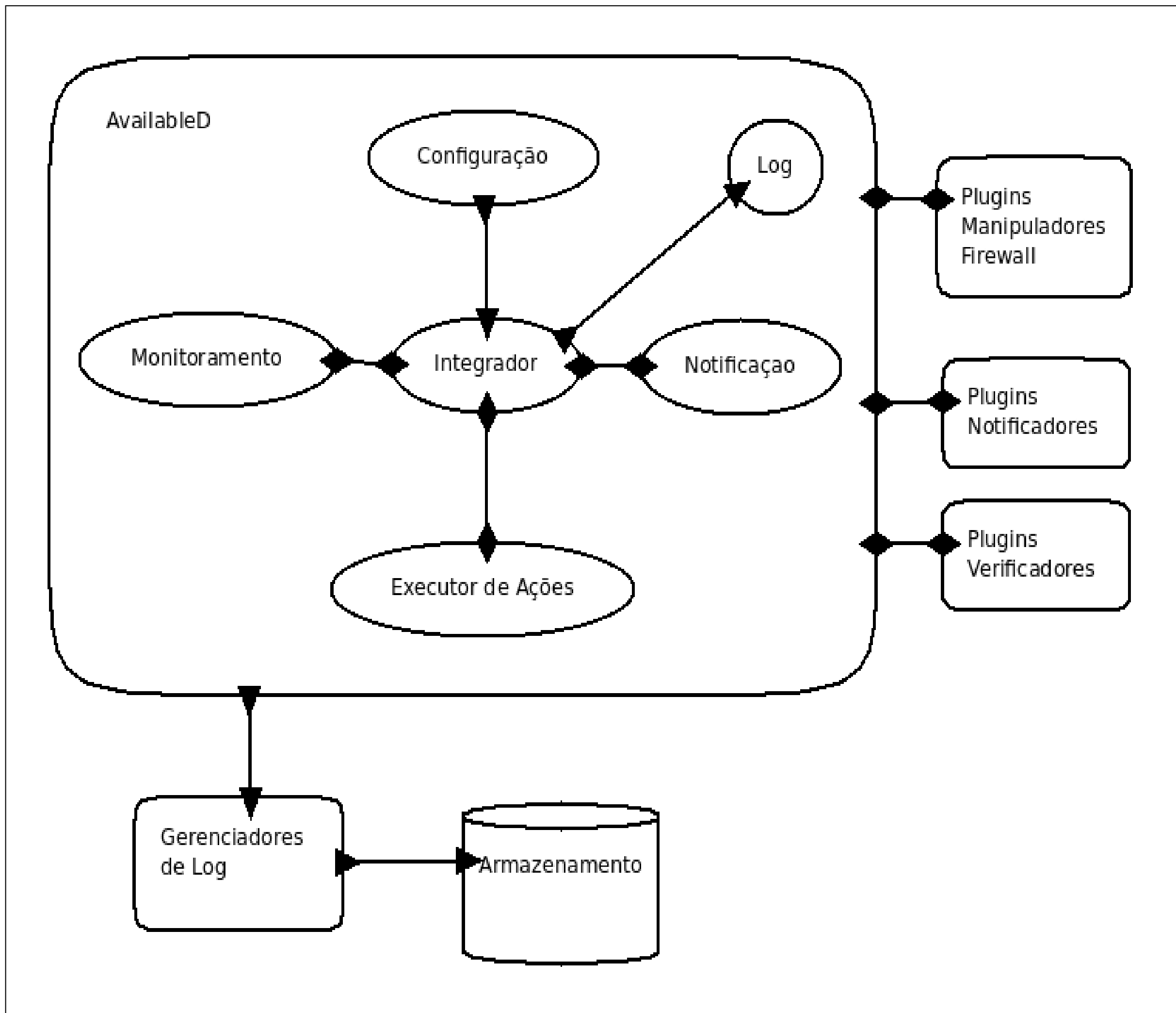
O módulo configuração é um componente crítico para o alcance das metas de simplicidade e facilidade de uso propostas pelo *AvailableD*, pois consiste na parte do sistema com a qual o usuário interage mais diretamente. Por consequência, será de considerável importância na avaliação de usabilidade.

Para modelar o formato do módulo de configuração, foram analisadas ferramentas com as quais os administradores de redes estão acostumados a utilizar e configurar. As observadas foram: PostgreSQL, MySQL, Dnsmasq, Apache2, PHP5, Bind, Samba, Postfix, OpenLDAP, DHCP3 Server e OpenSSH. Os aspectos observados foram 2:

1. **Forma como é realizada a configuração**: refere-se a interface tradicional oferecida aos usuários para informar os valores de configuração. Em todas as ferramentas analisadas constatou-se que a configuração é realizada por meio da edição de arquivos de texto.

2. **Aspectos comuns nas interfaces de configuração**: como todas as ferramentas utilizam arquivos textuais de configuração, o que foi considerado neste quesito são os aspectos mais comuns entre a sintaxe dos arquivos. Como resultado da análise, constatou-se que a grande maioria utiliza o formato *parâmetro valor1 valor2 valor3 ... valorN* (36%), ou então, *parâmetro=valor1,valor2,valor3,...,valorN* (55%) para a atribuição de valores aos

Figura 3.9 – Arquitetura do *AvailableD*, seus *plugins* e programas externos



Fonte: do autor

parâmetros de configuração. Nas ferramentas que utilizam componentes de bloco em sua configuração (apenas 3), o padrão mais presente (67%) foi o uso de abre e fecha chaves ({}) como delimitador de escopo. Os comentários nos arquivos, na grande maioria das ferramentas analisadas (91%), utilizam o caracter "#". Qualquer sequência de caracteres após o "#", até o final da linha, é interpretado como parte do comentário. Algumas ferramentas utilizam o ";", e outras suportam ainda comentários no estilo "/**/" e "//". Para delimitar parâmetros, o mais comumente utilizado é uma nova linha (64%).

A partir dos dados resultantes das análises realizadas, ficou evidente que uma alternativa aceitável e amplamente utilizada é a configuração através de arquivos textuais. Já para a definição dos parâmetros e valores de configuração uma das formas mais utilizadas e aceitas é o formato *parâmetro=valor1,valor2,valor3,...,valorN*. O sinal de igualdade passa a representar atribuição de valor e a vírgula passa a ser o separador quando há múltiplos valores para um

mesmo parâmetro. Como estilo de comentários foi escolhido somente o “#”. Optou-se por utilizar um “;” como delimitador de parâmetros, embora não seja muito utilizado. A razão é permitir que os valores para os parâmetros admitam nova linha sem comprometer a legibilidade.

Para permitir a configuração das funcionalidades necessárias pelo *AvailableD*, considerando sua arquitetura proposta, constatou-se a necessidade dos seguintes conjuntos de informações:

1. Lista de serviços a serem monitorados, incluindo os servidores que os prestam;
2. Comandos utilizados para realizar as verificações dos serviços;
3. Comandos do *firewall* para efetuarem a adição e remoção dos redirecionamentos realizados;
4. Uma forma de definir para cada servidor, quais são seus possíveis servidores substitutos, isto é, os servidores para os quais os redirecionamentos poderão ser realizados. Isso é importante, pois em algumas situações, servidores do mesmo serviço não estão preparados para atender requisições de outros servidores. Como exemplo, podemos considerar cenários mestre/escravo, onde é possível realizar redirecionamento do escravo para o mestre, porém o contrário não é possível, já que não é todo o tipo de solicitações que um servidor escravo pode atender;
5. Informações sobre os gerenciadores de *log*;
6. Informações dos *plugins* notificadores utilizados.

Para melhor estruturar e organizar as informações do arquivo de configuração, foram criados três escopos, o global, o serviço e o servidor. Eles possuem vínculo hierárquico, sendo o mais abrangente o global e o mais específico o servidor. O serviço engloba um ou mais servidores, representando as N réplicas do serviço.

1. **Global**: escopo onde são declarados todos os valores padrões globais, ou seja, válidos quando os parâmetros não estiverem localmente definidos nos escopos mais específicos;
2. **Serviço**: escopo para definir os serviços e suas informações. Os parâmetros declarados neste escopo constituem de valores padrões dentro do escopo, e sobrescrevem os valores globais;
3. **Servidor**: escopo utilizado para definir os servidores de cada serviço. Os parâmetros definidos neste escopo sobrescrevem os demais valores e são específicos para o servidor no qual foram declarados.

Na Figura 3.10 encontra-se a sintaxe básica do arquivo de configuração do *AvailableD*. As primeiras declarações de parâmetros são referentes a valores globais. Após, iniciam as definições de serviços, que são identificadas pela palavra reservada *service*. Depois da palavra *service*, vem o nome do serviço sendo declarado, que deve ser único. Entre o abre e fecha chaves conseguintes, fica o escopo do serviço declarado. Os valores para os parâmetros declarados neste escopo somente são válidos para ele e para os servidores declarados internamente. Para definir um servidor, utiliza-se a palavra reservada *server* e, após, informa-se o endereço IP do servidor, opcionalmente seguido da porta utilizada para acesso ao serviço. Entre o abre e fecha chaves posteriores fica o escopo do servidor, no qual os valores informados aos parâmetros são específicos para o servidor.

Figura 3.10 – Sintaxe básica do arquivo de configuração do *AvailableD*

```
parametro1 = valor1;
parametro2 = valor2;
...
service name1 {

    parametro1 = valor1;
    ...

    server ip1:porta1 {
         parametro1 = valor1;
         ...
    }
    server ip2:porta2 {
         parametro1 = valor1;
         ...
    }
    ...

}
service name2 ...
```

Fonte: do autor

As informações de configuração, na grande maioria, podem ser definidas em qualquer escopo. A escolha por esta flexibilidade na definição dos valores foi para permitir maior dinamismo e granularidade na configuração. Desta maneira é possível, por exemplo, manter uma configuração de notificação global, utilizada para todos os serviços e servidores, ao mesmo tempo que para um determinado serviço (ou servidor), a configuração pode ser alterada, através da redefinição dos valores dos parâmetros de notificação no escopo deste serviço (ou servidor). A redefinição de valores é realizada através da redeclaração dos parâmetros em escopos diferentes. Outro aspecto positivo desta convenção, é a possibilidade de, em poucas linhas, configurar o monitoramento de um grande número de máquinas, utilizando os escopos menos específicos para definição de informações comuns.

Para melhor entendimento da definição de valores padrões e momentos em que são usados, considere o arquivo de configuração da Figura 3.11. Para o servidor 10.10.10.1, o valor de *ar-*

quivo_log será “arquivo_servidor”, pois os valores informados no escopo servidor sobrescrevem todos os demais. No caso do servidor 10.10.10.2, o valor de *arquivo_log* será “arquivo_servico”, pois não há valor definido para o servidor e o escopo do serviço sobrescreve o global. Por fim, no caso do servidor 10.10.10.3, o parâmetro *arquivo_log* conterá o valor “arquivo_global”, pois não há nenhuma outra declaração mais específica.

Figura 3.11 – Exemplo de configuração com sobrescrita de valores dos parâmetros

```
arquivo_log = arquivo_global;

service dns {

  arquivo_log = arquivo_servico;

  server  10.10.10.1 {
     arquivo_log = arquivo_servidor;
  }

  server 10.10.10.2 {
  }

}

service smtp {
  server 10.10.10.3 {
  }
}
```

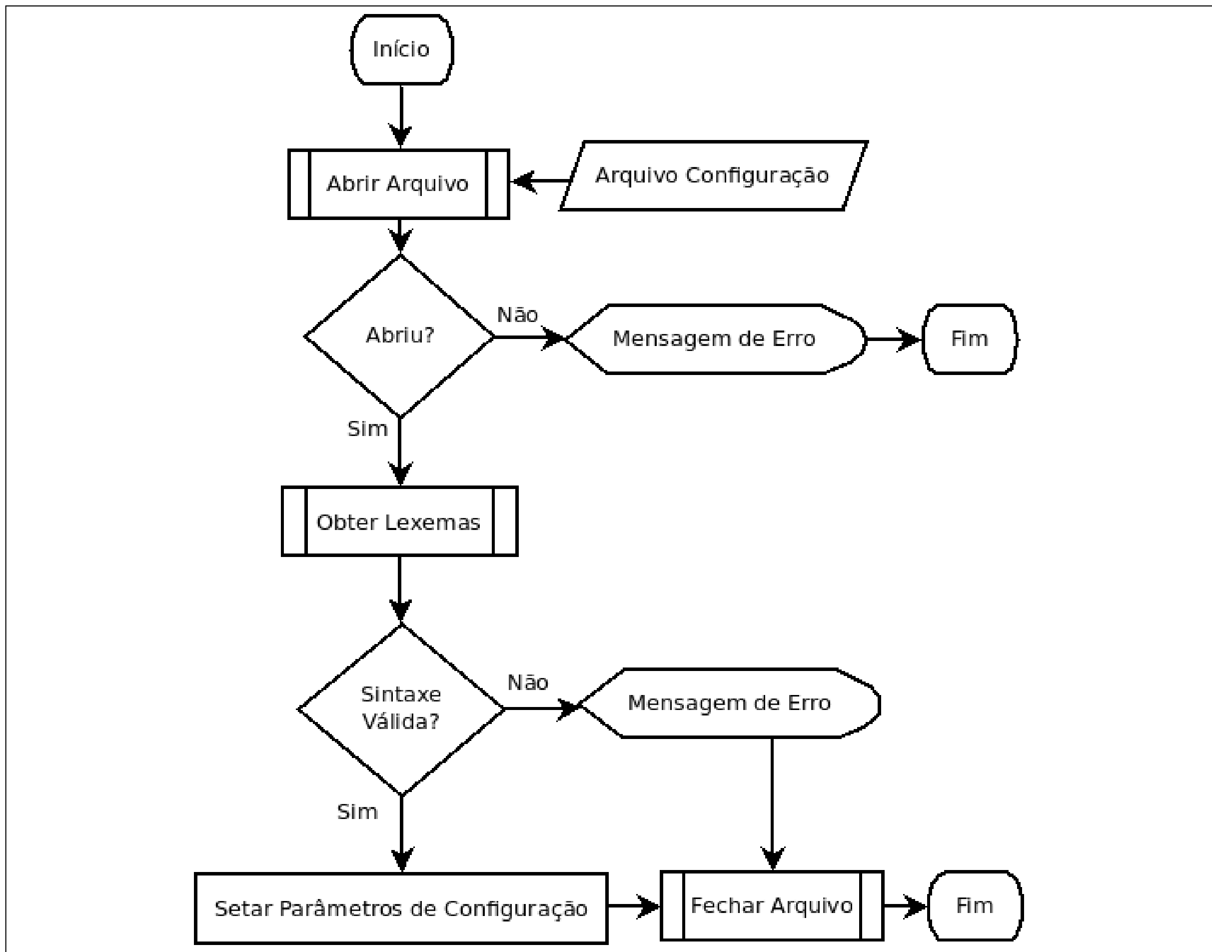
Fonte: do autor

Informações adicionais sobre o arquivo de configuração encontra-se no documento de projeto, presente no Apêndice G. Na Figura 3.12 está ilustrado o funcionamento do módulo na forma de fluxograma, onde é apresentada a maneira com que o arquivo de configuração é processado. Através da figura, percebe-se que o funcionamento do módulo consiste basicamente em:

1. Abrir o arquivo de configuração;
2. Obter seus lexemas (textos que correspondem aos elementos do arquivo);
3. Analisar a sintaxe;
4. Setar os parâmetros de configuração;
5. Exibir mensagem quando algum erro for diagnosticado;
6. Fechar o arquivo de configuração e parar a execução ao terminar de processá-lo.

Mais informações do módulo estão disponíveis no documento de projeto no Apêndice G.

Figura 3.12 – Fluxograma de execução do módulo configuração do *AvailableD*



Fonte: do autor

3.3.2 Módulo Monitoramento

Os principais componentes deste módulo são os verificadores de serviços. Internamente, o papel do módulo é executar os verificadores, receber e analisar suas respostas. Os verificadores são especificados no arquivo de configuração. Eles são descritos na forma de comandos a serem executados. As informações essenciais para a execução de praticamente qualquer verificador de serviços consistem no endereço de rede da máquina aonde o serviço é prestado e o número da porta para conexão com o serviço.

Para permitir a correta comunicação entre o *AvailableD* e seus *plugins* de verificação, foram definidas algumas convenções que devem ser seguidas pelos mesmos:

1. **Transmissão de Mensagens**: a transmissão de mensagens para o *AvailableD* dá-se pela impressão das mesmas na saída padrão do programa. A finalidade das mensagens é exclusivamente reportar problemas. Mensagens de sucesso, por exemplo, não são utilizadas pelo *AvailableD*, mas recomenda-se que sejam implementadas, pois constituem em uma

maneira de informar o que está acontecendo aos usuários, quando testarem os *plugins* fora do *AvailableD*;

2. **Identificação do Resultado da Verificação**: o resultado da verificação dá-se pelo código de saída do *plugin*, ficando definidos os seguintes valores:

   (a) **0**: a verificação foi realizada e o serviço está disponível;

   (b) **126** e **127**: valores reservados para identificar que a verificação não pode ser realizada. Uma ocasião de uso destes valores é se os argumentos passados para o *plugin* não forem suficientes para a verificação do serviço. Estes valores foram escolhidos porque são uma convenção de valores retornados pelo próprio *shell* (executor de programas do sistema operacional), significando (no caso do *shell*) que o comando não é executável, ou não foi encontrado, respectivamente;

   (c) **1**: a verificação foi realizada e o serviço encontra-se indisponível.

   As mensagens impressas somente serão utilizadas em caso de códigos de retorno diferentes de 0.

Dadas estas convenções, torna-se simples escrever ou modificar *plugins* existentes para o *AvailableD*. Estas convenções são semelhantes as do Nagios, o que permite que alguns de seus *plugins* sejam utilizados até mesmo sem necessidade de modificações. Na Figura 3.13 está ilustrado o funcionamento, de forma genérica, dos *plugins* verificadores. Através da figura, percebe-se que, em linhas gerais, o funcionamento destes *plugins* consiste em:

1. Verificar a conformidade dos argumentos passados na linha de comando;

2. Imprimir mensagem de erro e parar a execução retornando código 126 ou 127 em caso da não conformidade dos argumentos;

3. Verificar a disponibilidade do serviço;

4. Imprimir mensagem se o serviço estiver indisponível;

5. Parar a execução retornando código 0 se o serviço estiver disponível ou código 1 se o serviço não estiver.

A Figura 3.14 ilustra o fluxo de execução do módulo monitoramento. Nota-se que o funcionamento do módulo é muito simples, consistindo basicamente de três passos:

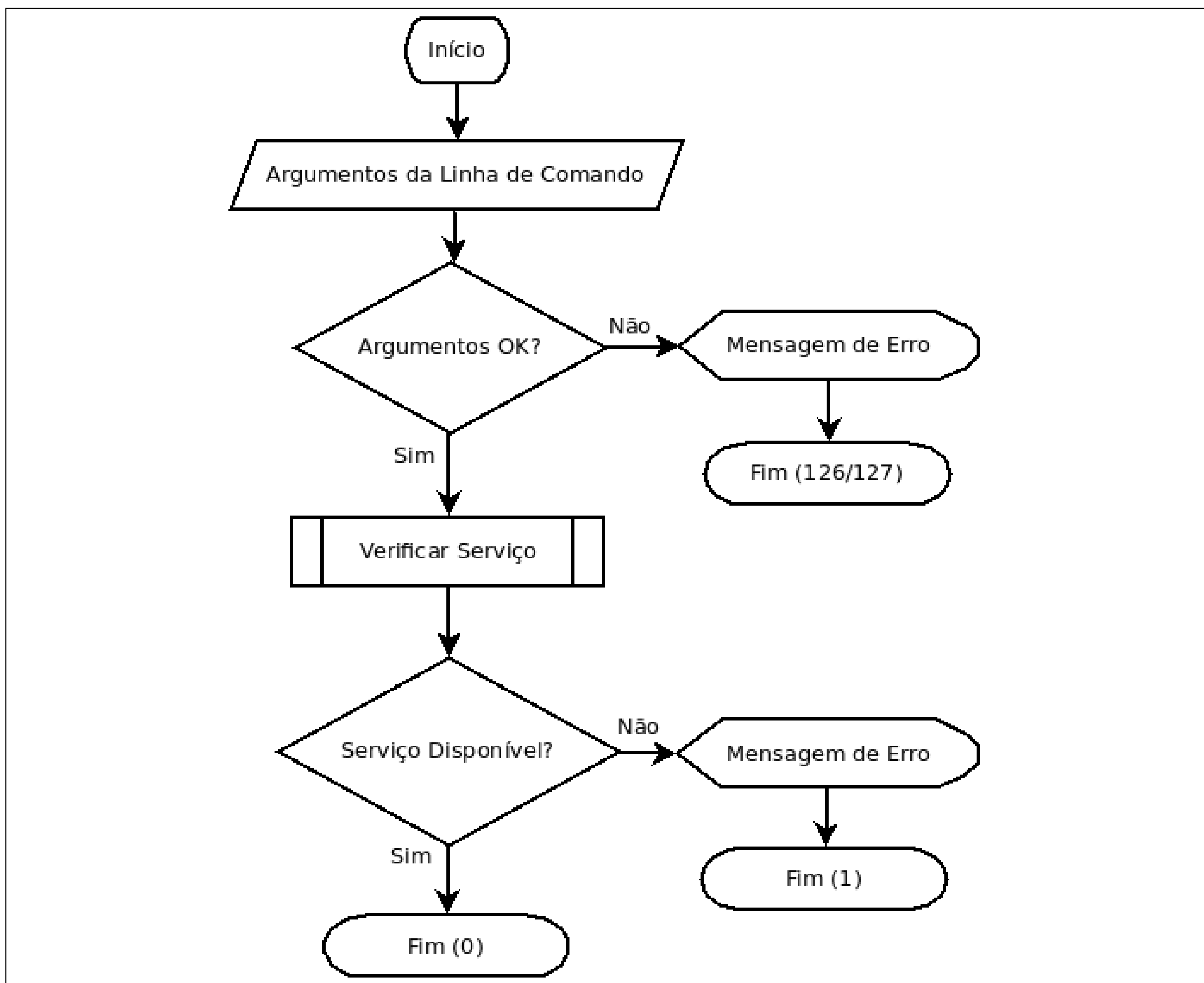
1. Executar o *plugin* de verificação da disponibilidade do serviço;

2. Receber e processar a resposta;

3. Terminar a execução retornando a resposta processada;

Maiores informações do módulo encontram-se no documento de projeto, Apêndice G.

Figura 3.13 – Funcionamento genérico dos *plugins* verificadores

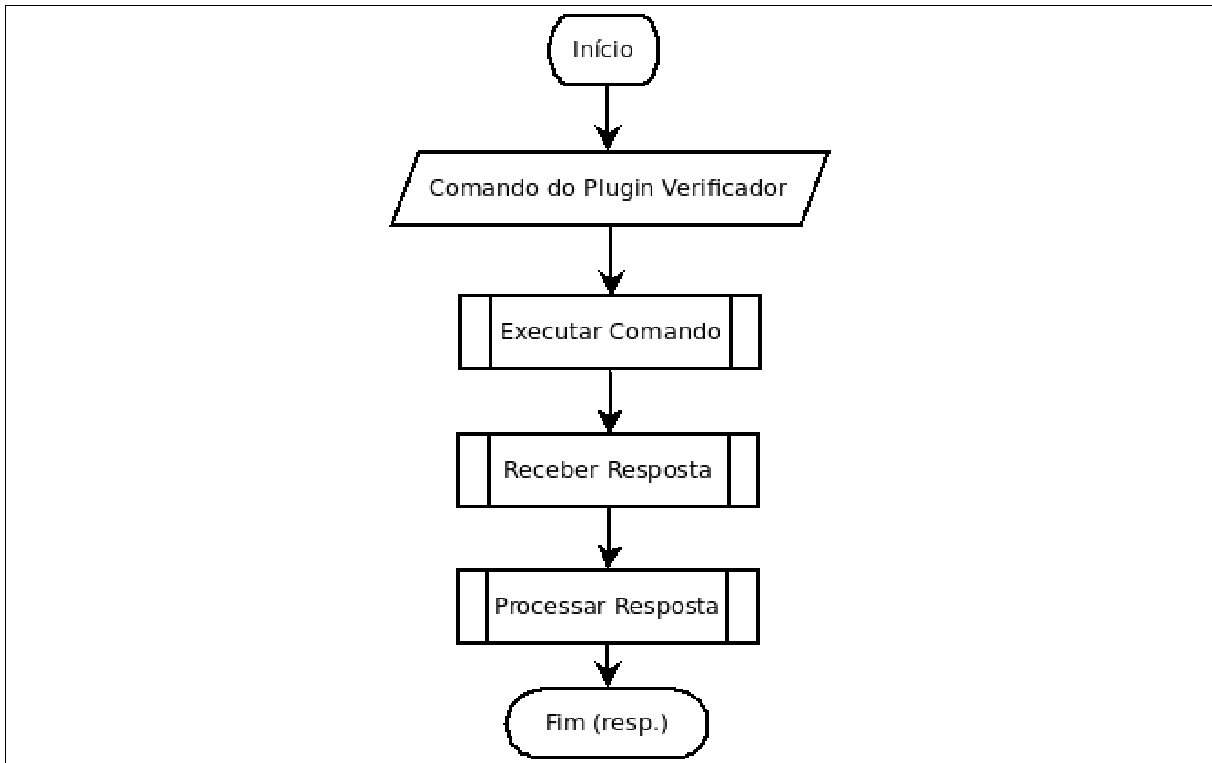


Fonte: do autor

3.3.3 Módulo Executor de Ações

Este é o módulo responsável por executar as ações definidas pelo usuário nos momentos de alteração da situação de disponibilidade dos servidores. As ações consistem nos comandos necessários para realizar e remover os redirecionamentos de pacotes. O funcionamento do módulo executor é similar ao módulo de monitoramento, pois sua estrutura é baseada na execução de comandos (*plugins*) definidos pelo usuário. Seu funcionamento está ilustrado na Figura 3.15 na forma de fluxograma de execução. Resumidamente, seu funcionamento consiste em:

Figura 3.14 – Funcionamento do módulo monitoramento do *AvailableD*



Fonte: do autor

1. Verificar a ocorrência de alteração na disponibilidade do serviço/servidor através dos resultados obtidos pelo módulo monitoramento;

2. Remover redirecionamento adicionado para outro servidor quando houver alteração no estado do servidor para disponível.

3. Quando houver alteração no estado do servidor para indisponível, remover os redirecionamentos de outros servidores para o servidor falho, redirecioná-los para novos substitutos, e adicionar redirecionamento do servidor falho para um substituto.

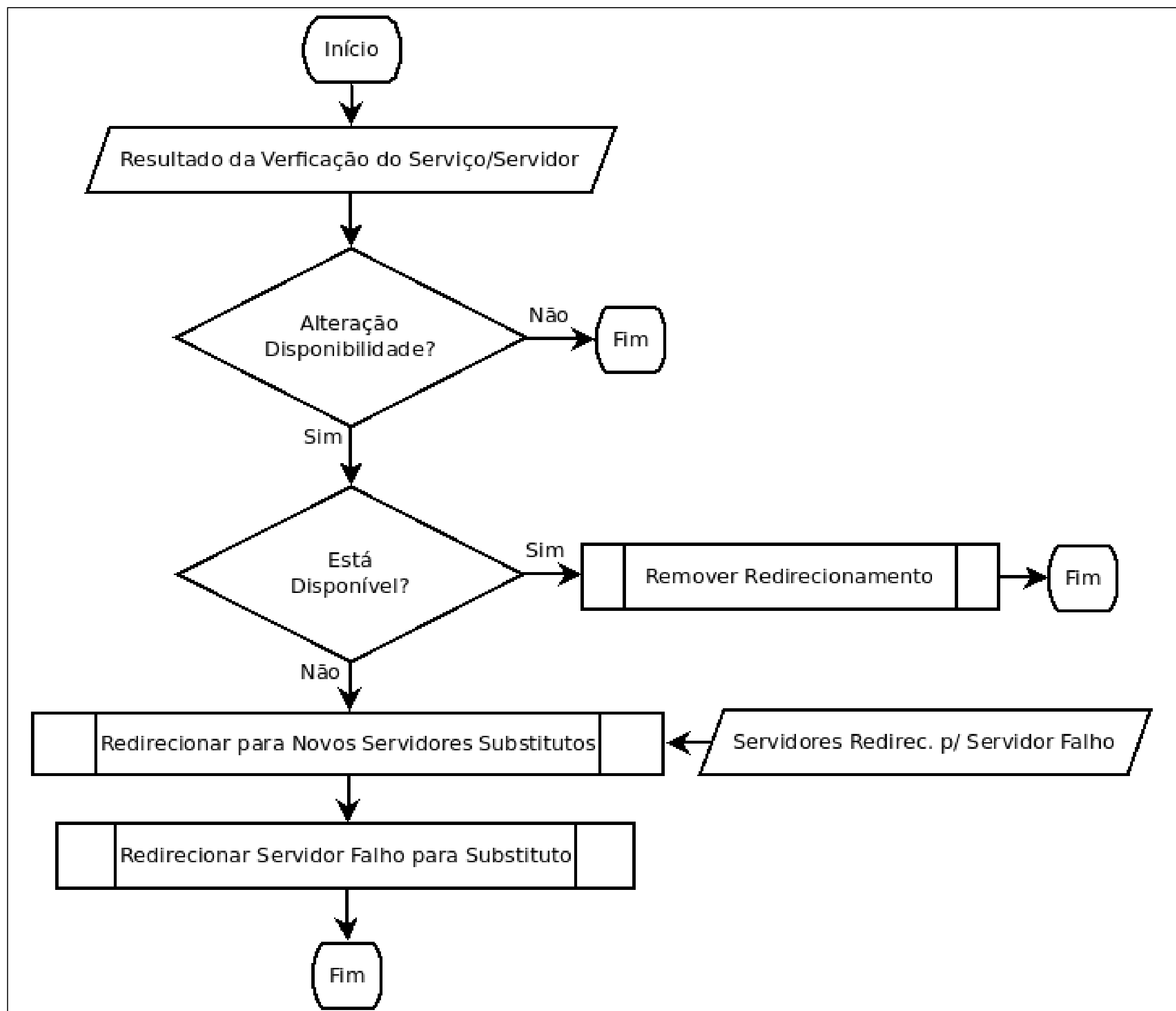
4. Parar a execução quando não houver alteração na disponbilidade e após adicionar/remover os redirecionamentos;

Para maiores informações do módulo, consulte o documento de projeto (Apêndice G).

3.3.4 Módulo Log

Para registro de informações relevantes de eventos e ações realizadas, principalmente pelos módulos de monitoramento e execução de ações, faz-se necessária a utilização de um sistema de *log*. Este módulo prevê a utilização de gerenciadores de *log*, o que torna dinâmicos a

Figura 3.15 – Funcionamento do módulo executor de ações do *AvailableD*



Fonte: do autor

manipulação e gerenciamento deste tipo de informação.

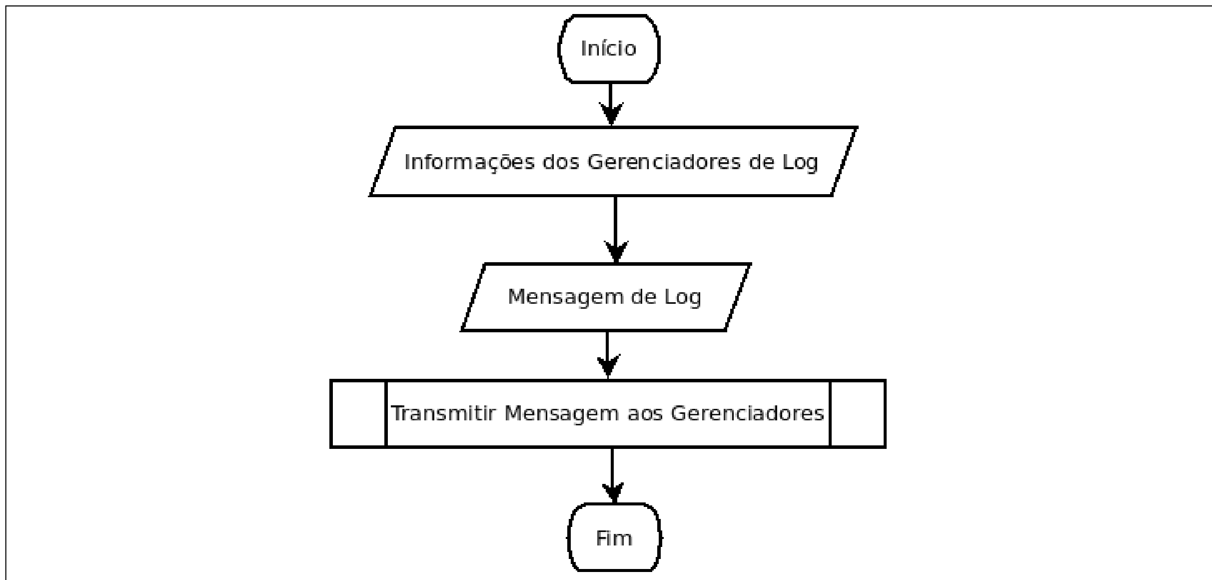
O papel do módulo consiste apenas em transmitir as mensagens de *log* para os gerenciadores que, de fato, realizam o armazenamento. Seu funcionamento é ilustrado na Figura 3.16 e resume-se em:

1. Receber a mensagem de *log* e transmití-la aos gerenciadores de *log*;

2. Terminar a execução.

Mais detalhes sobre o módulo encontram-se no documento de projeto, Apêndice G.

Figura 3.16 – Funcionamento do módulo log do *AvailableD*



Fonte: do autor

3.3.5 Módulo Notificação

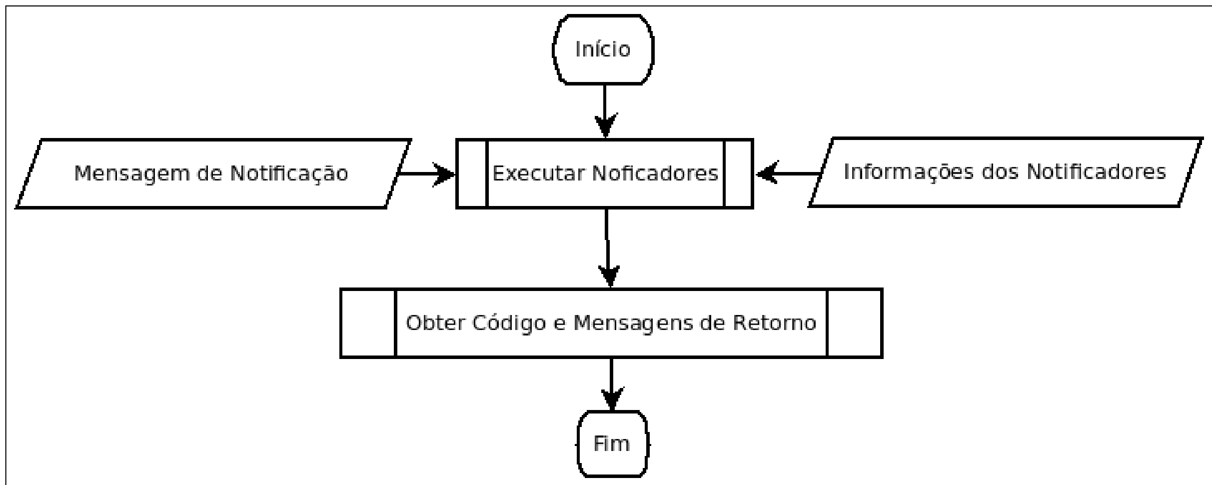
Este módulo prevê a utilização de *plugins*, o que provê flexibilidade e extensibilidade, criando a possibilidade dos usuários desenvolverem ou adaptarem *plugins*, a fim de melhor adaptação às suas necessidades. A convenção para o desenvolvimento dos *plugins* de notificação segue as mesmas especificações dos *plugins* para verificação de serviços.

O módulo possui a incubência de executar os *plugins* de notificação, passando a eles a mensagem a ser notificada, e aguardar o retorno de sucesso ou não na execução das notificações. Em caso de falhas na realização das notificações, as mensagens retornadas pelos *plugins* são importantes para identificação de possíveis problemas de configuração ou mau funcionamento dos notificadores, portanto, também devem ser obtidas. O funcionamento do módulo é ilustrado na Figura 3.17 e consiste em:

1. Receber a mensagem a ser notificada e executar os notificadores, passando a mensagem a eles;
2. Capturar o código de saída e as mensagens emitidas pelos notificadores;
3. Terminar a execução.

Maiores informações sobre o módulo são encontradas no documento de projeto, Apêndice G.

Figura 3.17 – Funcionamento do módulo notificação do *AvailableD*



Fonte: do autor

3.3.6 Módulo Integrador

É o responsável por realizar a integração entre os demais módulos. A integração consiste, basicamente, em realizar chamadas às funcionalidades dos módulos e intermediar a troca de informações entre eles, aplicando a lógica de operação necessária para obter o funcionamento previsto, e fazer com que atuem como um único organismo.

No algoritmo da Figura 3.18 está representada de forma sucinta a lógica de integração do *AvailableD*. Em resumo, realiza-se a obtenção das informações de configuração, as quais contém os serviços e servidores a serem monitorados, e informações pertinentes, utilizadas pelos módulos. O próximo passo é iniciar um *thread* (fluxo de execução) para cada servidor de cada serviço, permitindo que sejam monitorados de forma paralela. Para o monitoramento, são passadas informações de configuração, que contém, dentre outras coisas o *plugin* verificador. As informações resultantes do monitoramento são encaminhadas ao executor de ações, assim como os dados de configuração, que contém os *plugins* que manipulam as regras de roteamento. O executor de ações irá determinar a necessidade ou não de realizar/remover redirecionamentos. Quando retornar alguma mensagem, será realizado *log* e notificação da mesma.

Figura 3.18 – Lógica de integração entre os módulos do *AvailableD*

```
1: conf ← configuracao();
2: for all servico ∈ conf.servicos do
3:    for all servidor ∈ servico.servidores do

4:       thread(

5:       for SEMPRE do
6:          (resultado, mensagem) ← monitoramento(servidor.conf);
7:          if mensagem then
8:             log(mensagem);
9:          end if
10:         mensagem ← executorAcoes(resultado, servidor.conf);
11:         if mensagem then
12:            log(mensagem);
13:            notificacao(mensagem, servidor.conf);
14:         end if
15:      end for

16:      ); {final do código da thread}

17:   end for
18: end for
```

Fonte: do autor

4 Tecnologias e Implementação

Antes de ter sido dado início a fase de implementação, o sistema foi especificado e modelado. Como resultado, foi criado o documento de projeto da solução, que encontra-se no Apêndice G. Este documento serviu de guia durante toda a etapa de desenvolvimento do *AvailableD* e as especificações presentes nele condizem, na maior parte, com o que foi de fato implementado.

A solução tem por objetivo ser *open source* (de código aberto), disponibilizada sob os termos da licença *Gnu General Public Licence version 3* (GPLv3), a fim de possibilitar a liberdade de ser executada, adaptada e redistribuída, sem nenhum custo, por qualquer pesssoa. Outro fator levado em consideração na escolha desta licença, é o grande número de ferramentas que são disponibilizadas sob os mesmos termos, ou compatíves (versões anteriores da licença), o que possibilita que sejam utilizadas no desenvolvimento da solução.

Os sistemas operacionais suportados nesta primeira versão da solução são apenas sistemas GNU/Linux.

4.1 Tecnologias de Apoio

Para a construção do *AvailableD*, foram utilizadas várias tecnologias e soluções já consolidadas. As escolhas de tais recursos foram baseadas em um conjunto de fatores, visando possibilitar que os recursos mais adequados fossem selecionados. Em linhas gerais, os aspectos mais relevantes analisados foram:

1. **Adequação ao problema**: refere-se ao quão adequada a tecnologia é para o problema que deseja-se resolver. Este item é influenciado por uma gama de detalhes, que vão além de simplesmente analisar se uma dada solução resolve determinado problema. A análise deve ser mais específica e levar em consideração fatores como integração com outras tecnologias, impactos positivos e negativos que seu uso causaria no sistema como um todo, dependências existentes, requisitos de desempenho e de confiabilidade, dentre outros;

2. **Consolidação**: procura-se utilizar soluções e tecnologias já consolidadas e bastante utilizadas, pois é um dos maiores indicativos de uma garantia maior de qualidade;

3. **Condições de uso**: trata-se das restrições de utilização impostas pelos seus proprietários.

Por tratar-se de uma solução livre, descarta-se a possibilidade de utilização de ferramentas proprietárias ou possuidoras de licenças incompatíveis com a GPLv3;

4. **Conhecimentos dos Desenvolvedores do *AvailableD***: é desejável que os desenvolvedores possuam conhecimento prévio quanto a utilização das tecnologias/soluções, pois isso, além de reduzir o tempo de desenvolvimento, no geral, possibilita uma utilização mais adequada e eficiente, devido a conhecimentos específicos, adquiridos com a experiência.

Tendo definido o escopo do *AvailableD* e suas problemáticas inerentes, foi realizada uma pesquisa para determinar as tecnologias e ferramentas auxiliares a serem utilizadas, tomando como critérios de avaliação os aspectos supracitados. As seguintes tecnologias foram então selecionadas:

1. **Linguagem de programação C**: escolhida como principal linguagem para o desenvolvimento do *AvailableD*. Foi selecionada por ser uma linguagem já consolidada, com ampla documentação e de maior conhecimento dos desenvolvedores da solução.

2. **Flex** (`http://flex.sourceforge.net/`): é uma ferramenta *open source* que gera código em C para a análise léxica. É uma ferramenta já consolidada, com uma documentação bastante clara e completa. O código gerado é otimizado através da utilização de matrizes de estado para identificação dos *tokens* (classes de elementos da gramática), e não possui dependências além de bibliotecas padrões da linguagem C. É importante salientar que já foi utilizada anteriormente pelos desenvolvedores do *AvailableD*. O *Flex* é utilizado no módulo configuração, para realizar a análise léxica do arquivo de configuração e auxiliar na análise sintática;

3. **MSMTP** (`http://msmtp.sourceforge.net/`): consiste em um cliente para envio de emails. Possui suporte a vários métodos de autenticação e, caso esteja disponível no sistema a biblioteca *GnuTLS* (`http://www.gnu.org/s/gnutls/`) ou a *OpenSSL* (`http://www.openssl.org/`), provê suporte a utilização de criptografia TLS na conexão com o servidor de email. Esta solução é livre e portável para vários sistemas operacionais. O MSMTP possui uma biblioteca SMTP escrita em C com várias funcionalidades e fácil de ser utilizada, a qual pode ser facilmente adaptada para implementar um componente para notificação por envio de email. Um aspecto positivo é a boa documentação encontrada e uma comunidade de desenvolvimento ativa;

4. **Mon** (`https://mon.wiki.kernel.org/`): esta é uma solução para monitoramento de serviços. O que foi aproveitado desta solução, são alguns de seus verificadores de serviço,

que foram reescritos e adaptados para utilização com o *AvailableD*. A justificativa de utilizar estes verificadores é a simplicidade com que foram escritos.

5. **Autotools** (`http://www.gnu.org/software/autoconf/`): conjunto de ferramentas que auxilia e automatiza o processo de configuração e instalação. Esta ferramenta gera *scripts* de instalação capazes de verificar o ambiente em que são executados em busca das dependências do *software* como, por exemplo, bibliotecas, programas necessários para compilação e instalação, etc.

4.2 Arquitetura Concretizada na Implementação

O protótipo desenvolvido seguiu as convenções definidas no projeto, materializando a arquitetura proposta e implementando as funcionalidades previstas. A seguir, são apresentados os detalhes de implementação de cada módulo do sistema.

4.2.1 Configuração

A configuração do *AvailableD* é realizada através de um único arquivo localizado em /etc/availabled/availabled.conf. A implementação da análise léxica e sintática do arquivo foi realizada com o auxílio do *Flex*. Para a análise léxica, basta informar as expressões regulares (padrões textuais) de cada *token* (tipo de símbolo). Para analisar a sintaxe, foi utilizada a funcionalidade do *Flex* chamada *start condition* (condição de início), que permite definir o momento em que cada padrão passa a estar ativo, ou seja, sua ocorrência passa a ser esperada.

Na Figura 4.1 são exibidas algumas das expressões regulares utilizadas. Na coluna da esquerda ficam os nomes (apelidos) atribuídos às expressões regulares presentes na coluna da direita. Na Figura 4.2 encontram-se as condições de início utilizadas, definidas pelo comando "%s" do *Flex* e declaradas lado a lado.

Figura 4.1 – Algumas das expressões regulares utilizadas na análise léxica

```
espace                      [ \t\n]+
parameter                   [a-z][a-z0-9_]{0,24}
value                       ([^ ,\n\t;"]*)|(\"(([^\"\n])|(\\\")|(\\\n))*\")
value_separator             ,
delimiter                   ;
service_name                [a-z][a-z0-9_]{0,29}
ip_address                  [a-fA-F0-9.:]{1,45}
port                        [0-9]{1,5}
server_identification       ({ip_address}(:{port})?)|(\[{ip_address}\](:{port})?)
comment                     #[^\n]*$
```

Fonte: do autor

Figura 4.2 – Condições de início utilizadas para a análise sintática

```
%s ASSIGN VALUE SERVICE_NAME SERVER_IDENT OPEN_KEY SERVICE_SCOPE SERVER_SCOPE DELIMITER
```

Fonte: do autor

O processamento para cada *token* é realizado no momento em que o mesmo é encontrado. Para entender a maneira como é realizada a análise e processamento do arquivo de configuração, tome como exemplo a palavra reservada "service", que identifica o início da declaração de um serviço a ser monitorado. O processamento deste *token* consiste apenas em gravar que o último *token* encontrado foi o "service" e iniciar o padrão que define o nome do serviço, o que é esperado após encontrar este *token*.

Na Figura 4.3 está o código que realiza o processamento o *token* "service". A declaração `<INITIAL>"service"` significa que a expressão regular "service" tem como condição de início *INITIAL*, que é uma condição definida pelo *Flex* e é a primeira a estar ativa. A diretiva *#ifdef DEBUG_MODE* serve apenas para imprimir na tela que foi encontrado o *token* "service", quando o programa for compilado em modo de *debug* (depuração). A variável *cf_iLastToken* é responsável por armazenar o último *token* encontrado. O comando *BEGIN*, é um comando do *Flex* que ativa uma condição de início, isto é, faz com que os padrões condicionados por ela sejam ativados, enquanto que os demais ficam inativos.

Figura 4.3 – Código que processa a palavra reservada "service"

```
<INITIAL>"service"            {

                                #ifdef DEBUG_MODE
                                  printf("SERVICE\n");
                                #endif

                                /* Update state */
                                cf_iLastToken = ABLETK_SERVICE;
                                BEGIN(SERVICE_NAME);

                              }
```

Fonte: do autor

4.2.2 Monitoramento de Serviços

O *AvailableD* é capaz de verificar a disponibilidade de serviços de rede. Qualquer serviço capaz de ser monitorado por *software* pode ser monitorado pelo *AvailableD*, bastando o desenvolvimento de um *plugin* que o verifique. A localização dos *plugins* no sistema fica em */usr/lib/availabled/plugins/check/*, porém, pode-se configurar para utilizar *plugins* presentes em qualquer outro lugar no sistema.

É possível configurar um tempo máximo que cada verificação vai esperar até determinar

que o serviço está indisponível. Isso evita que verificações demorem períodos de tempo muito grandes, mesmo quando os *plugins* permitem longos períodos de espera nas verificações.

Como cada servidor possui um endereço diferente, e por vezes a porta também, isto se tornaria um empecilho para definições globais de comandos de checagem, o que forçaria a definição dos mesmo nos escopos de cada servidor. Para resolver este problema, ficou definido que é possível utilizar duas macros (variáveis) nos comandos de verificação, que serão substituídas pelos valores específicos a cada servidor. Esta macros são:

1. *$SERVER*: representa o valor do endereço IP do servidor;
2. *$PORT*: representa o valor da porta de acesso ao serviço.

Para exemplificar a utilização destas macros, considere o seguinte comando responsável por monitorar um servidor HTTP: "http.monitor --path index.html --port $PORT $SERVER". Neste exemplo, *http.monitor* é o verificador, e recebe como argumentos o caminho de um arquivo a ser verificado (index.html), a porta para conexão ($PORT) e o endereço do servidor ($SERVER). Supondo que o servidor a ser monitorado possuísse o endereço IP 10.1.1.1 e utilizasse a porta 80 para conexão ao serviço HTTP, após as variáveis serem interpretadas, o comando ficaria: "http.monitor --path index.html --port 80 10.1.1.1".

A substituição das variáveis pelos seus valores, no caso do módulo monitoramento, é realizada apenas uma vez, no momento de inicialização do módulo para cada servidor sendo monitorado, pois as informações (endereço IP e porta) não mudam a cada verificação realizada. Na Figura 4.4 está o código responsável por interpretar as variáveis. O comando de verificação fica armazenado no atributo *sCheckCommand* da estrutura de informações do serviço monitorado (*opServiceInfo*). Através da utilização da função *able_strReplace*, as macros são substituídas pelos seus respectivos valores.

Para este primeiro protótipo, foram implementados três *plugins* verificadores de disponibilidade, os quais foram adaptados de *plugins* do *mon*. Os *plugins* implementados e seus detalhes estão descritos abaixo:

1. **dns-query.monitor**: verifica a disponibilidade de servidores DNS através da realização de consultas. A sinopse do comando é:

   dns-query.monitor [--tcp] [--record-type tipo_registro] --name domínio servidor,

   onde:

   (a) *--tcp*: se informado, será utilizado o protocolo TCP na requisição em vez do UDP;

Figura 4.4 – Código que interpreta as variáveis do módulo monitoramento

```
void able_checkerStart(able_svcinfo_t* opServiceInfo) {

  char* sTmp; /* Temporary string */

  /* ===> Process variables that will not change <=== */

  sTmp = able_integer2string(opServiceInfo->iPort);
  /* Process variables that will not change */
  opServiceInfo->sCheckCommand = able_strReplace("$SERVER", opServiceInfo->sIP,
                                                  opServiceInfo->sCheckCommand,
                                                  0, true
                                                 );
  opServiceInfo->sCheckCommand = able_strReplace("$PORT", sTmp,
                                                  opServiceInfo->sCheckCommand,
                                                  0, true
                                                 );
  /* Clean */
  free(sTmp);

}
```

Fonte: do autor

(b) *--name domínio*: informa o nome de domínio para o qual será realizada a consulta DNS;

(c) *--record-type tipo_registro*: determina que um tipo específico de registro DNS é esperado como resposta do servidor;

(d) *servidor*: endereço do servidor DNS para o qual será realizada a consulta.

2. **smtp.monitor**: testa o serviço SMTP, executando alguns comandos do protocolo. Sua sinopse é:

   smtp.monitor [--helo-domain domínio] [--port porta] servidor,

   onde:

   (a) *--helo-domain*: domínio utilizado como identificação para o servidor SMTP;

   (b) *--port porta*: porta a ser utilizada para conexão com o servidor. Se não informada, assume o valor 25;

   (c) *servidor*: endereço do servidor SMTP a ser verificado.

3. **http.monitor**: testa a disponibilidade de servidores HTTP realizando requisições a eles e, opcionalmente, analisando o conteúdo das páginas *web* retornadas. Possui a seguinte sinopse:

   http.monitor [--path caminho_pagina] [--match-regex regex] [--port porta] servidor,

   onde:

   (a) *--path caminho_pagina*: caminho da página web a ser verificada. Se não informado, assume a raiz, o que vai retornar a página *index*.

(b) *--match-regex regex*: expressão regular de um conteúdo a ser verificado na página;

(c) *--port porta*: porta a ser utilizada para conexão com o servidor. Se não informada, assume o valor 80;

(d) *servidor*: endereço IP do servidor HTTP.

Na Figura 4.5 está um trecho do código do *plugin* smtp.monitor (escrito em PERL). Neste trecho de código é realizada a conexão ao servidor SMTP e a análise dos códigos enviados como resposta pelo servidor. A primeira resposta analisada é a saudação emitida, que, se possuir código 220, significa aceitação por parte do servidor. Após receber uma saudação positiva (código 220), executa-se o comando SMTP HELO, que serve como início de conversação com o servidor. A resposta positiva para este comando consiste no código 250. Quando uma resposta negativa for recebida, executa-se o comando QUIT, que fecha a conexão. Após, seta-se a mensagem de erro (`$result->{"error"}`), pois o servidor será considerado indisponível.

4.2.3 Executor de Ações de Resposta

O *AvailableD* permite executar ações em resposta a modificação da situação de disponibilidade dos serviços. Estas ações são destinadas a adição e remoção dos redirecionamentos no sistema, porém, há a possibilidade de executar qualquer outro tipo de ação. As ações são na verdade os comandos (*plugins*) que serão executados no momento que um serviço tornar-se indisponível, e no momento que ele voltar a disponibilidade. Como o enfoque do *AvailableD* é realizar o redirecionamento de pacotes entre os servidores de um mesmo serviço, ele possibilita a configuração, por parte dos usuários, dos possíveis servidores substitutos para cada servidor. Dessa forma, só vai executar os comandos quando houver a definição de servidores substitutos e pelo menos um deles encontrar-se disponível.

Este módulo captura, além da saída padrão, a saída de erros dos programas, com a finalidade de permitir que interfaces de configuração de *firewalls* possam ser utilizadas diretamente como *plugins*, sem necessidade de *wrappers* (adaptadores). A necessidade de capturar a saída de erro, consiste em capturar as mensagens de erros emitidas por tais ferramentas, pois as mesmas não são enviadas à saída padrão, e sem estas mensagens, torna-se difícil diagnosticar o que está causando os erros. Desta maneira, o *firewall* mais comum em ambientes GNU/Linux, o IPTables, pode ser manipulado pelo *AvailableD* através de sua interface de configuração padrão (o próprio comando iptables).

Os redirecionamentos são sempre realizados de um servidor para outro. As informações básicas necessárias são endereço IP e número da porta. Estas informações variam de servidor

Figura 4.5 – Trecho do código do *plugin* smtp.monitor

```
if (!OpenSocket($server, $port)) {

    $result->{"error"} = "Unable to create SMTP connection to '$server' on port '$port'.";
    return $result;

  }

  # GREETINGS
  my $in                = <S>;
  $result->{"response"} .= $in;
  while ($in =~ /^220-/) { # Multiline response

    $in                  = <S>;
    $result->{"response"} .= $in;

  }
  if ($in !~ /^220 /) {

    print S "QUIT\r\n";
    close (S);

    $result->{"error"} = "Did not receive OK (220) greeting from $server. Not welcomed in server.";
    return $result;

  }

  # HELO
  print S "HELO $HELO_DOMAIN\r\n";
  $in                   = <S>;
  $result->{"response"} .= $in;
  while ($in =~ /^250-/) { # Multiline response

    $in = <S>;
    $result->{"response"} .= $in;

  }
  if ($in !~ /^250 /) {

    print S "QUIT\r\n";
    close (S);
    $result->{"error"} = "Did not get OK (250) response to HELO from $server.";
    return $result;

  }
```

Fonte: do autor

para servidor, sendo assim, foram definidas macros, de maneira semelhante ao módulo monitoramento. A diferença, no caso deste módulo, é que as informações não são de um único servidor, são de dois, um que apresenta/apresentou problemas e o outro é o substituto. As seguintes variáveis podem ser utilizadas nos comandos de *firewall*:

1. *$BROKEN_SERVER*: será substituída pelo endereço IP do servidor que estiver indisponível, no caso da adição de redirecionamentos, e pelo endereço do servidor que estava anteriormente indisponível, tratando-se dos comandos de remoção dos redirecionamentos;

2. *$BROKEN_PORT*: possui as mesmas definições de $BROKEN_SERVER, porém, para a porta;

3. *$ACTIVE_SERVER*: será substituída pelo endereço IP do servidor escolhido como subs-

tituto de um servidor falho, para o caso da adição de redirecionamentos. No caso da remoção, representa o endereço do servidor para o qual o redirecionamento havia sido realizado;

4. *$ACTIVE_PORT*: possui as mesmas definições de $ACTIVE_SERVER, porém, para a porta.

O módulo executor de ações processa suas variáveis em dois momentos: na inicialização do módulo para cada servidor sendo monitorado, e toda a vez que for necessário executar os comandos para adição ou remoção de redirecionamentos. As macros interpretadas no momento de inicialização são as iniciadas por $BROKEN, que correspondem as informações do próprio servidor monitorado, e que não sofrem alteração com o decorrer do tempo. As macros iniciadas por $ACTIVE somente podem ser substituídas imediatamente antes dos comandos serem executados, momento em que já foi escolhido um servidor substituto e as informações representadas pelas variáveis já são conhecidas.

Este módulo possui a peculiaridade de não poder ser executado de forma paralela entre *threads* de um mesmo serviço. O motivo é simples: existe uma condição de corrida intrínseca. Para melhor entendimento, vamos considerar três servidores de um mesmo serviço: A, B e C. Suponhamos que o servidor A falhou e foi escolhido o servidor B como substituto. A seguir, considere que o servidor B também falhou. A ação a ser tomada então, é redirecionar os pacotes com destino a B, e a A, para o último servidor disponível C. Neste mesmo momento, pode acontecer de o servidor A voltar a estar disponível e então ser realizada a remoção do redirecionamento para B. Neste caso, pode acontecer do redirecionamento A para B ser removido por duas vezes, o que geraria uma condição de erro. Além disso, o redirecionamento A para C será executado. Isso faria com que esta regra ficasse ativa e, mesmo o servidor A estando disponível, as requisições iriam para o servidor C. A função implementada que trata os redirecionamentos para servidores que tornaram-se indisponíveis está na Figura 4.6. Esta função obtém os servidores que estão redirecionados para um servidor que tornou-se indisponível, remove seus redirecionamentos, e tenta redirecionar para novos servidores substitutos. Como esta função é o ponto problemático, para tratar a condição de corrida basta executá-la após bloquear um *mutex* (mecanismo que fornece exclusão mútua) que impede a execução da função pelos *threads* que manipulam os redirecionamentos entre servidores de um mesmo serviço. Para melhor compreensão, a condição de corrida descrita está ilustrada na Figura 4.7.

Figura 4.6 – Função que trata redirecionamentos para servidores indisponíveis

```
void able_manageRedirectionsToBrokenServer(able_svcinfo_t* opBroken) {

  able_list_t* opListSvrRd;
  able_svcinfo_t* opSvrRdToBroken;
  able_svcinfo_t *opSvrSubstitute;

  opListSvrRd = able_searchServiceInfoByServerRedirectedTo(opBroken,
                                                          ((able_service_t*)opBroken->opService)->opInfo
                                                         );
  for (;able_listIsElement(opListSvrRd); opListSvrRd = opListSvrRd->opNext) {

    opSvrRdToBroken = (able_svcinfo_t*)opListSvrRd->opData;

    /* Undo redirection */
    able_executorRun(ABLEEXEC_UNDO_ACTION, opSvrRdToBroken, opBroken);

    /* Try redirect to a new substitute server */
    able_executorAction(opSvrRdToBroken);

  } /* End while (able_listIsElement(opListSvrRd)) */

  able_listFree(opListSvrRd, NULL); /* NULL as callback function to don't free data (able_svcinfo_t). */

  return;

}
```

Fonte: do autor

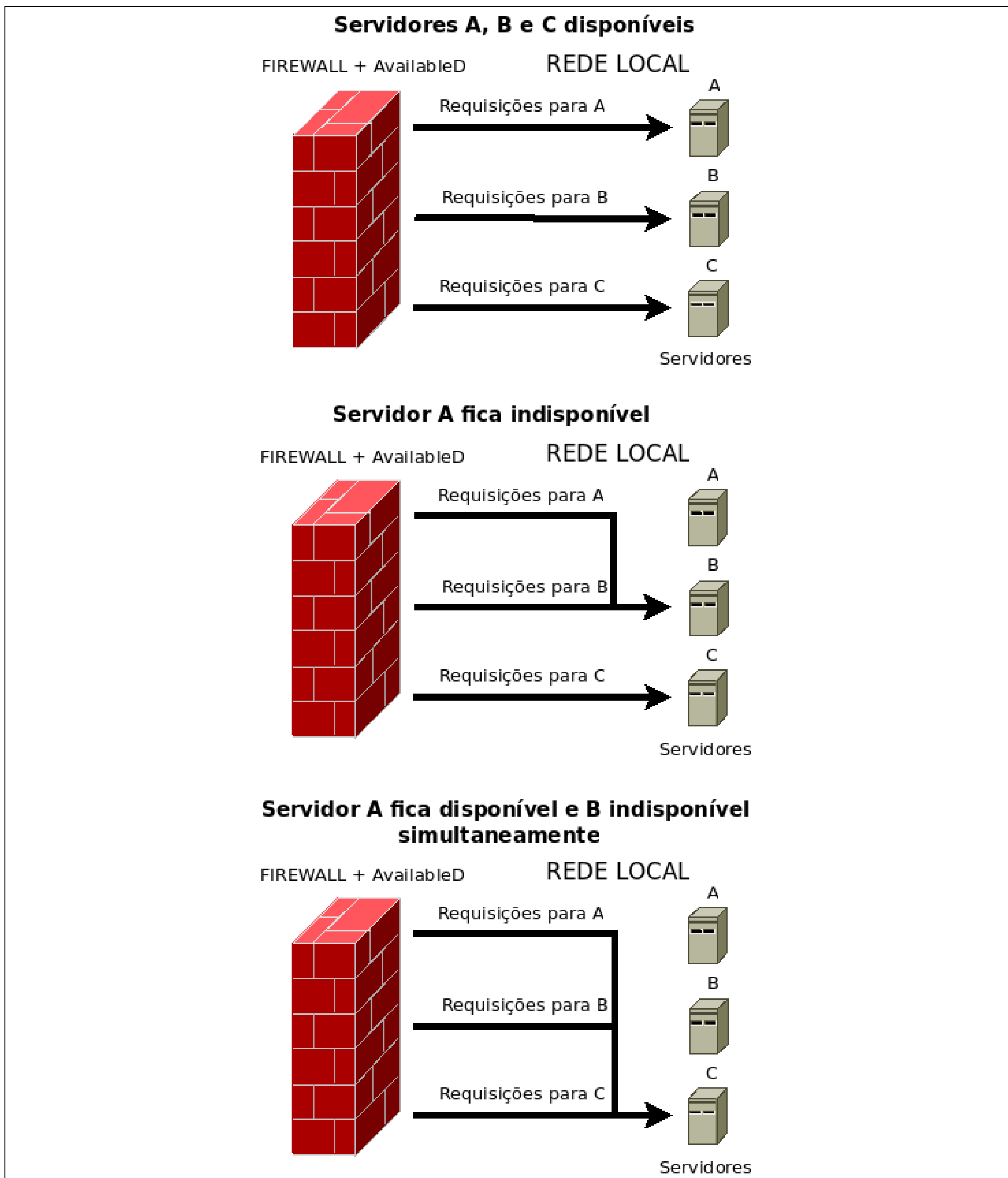
4.2.4 Log

Todas as informações relevantes, como detecção de falhas, adição e remoção de redirecionamentos e situações de erro encontradas serão enviadas para gerenciadores de *log*, a fim de serem armazenadas e posteriormente analisadas, se necessário. No desenvolvimento do primeiro protótipo, foi implementado um gerenciador de *log* que escreve em arquivos de *log* especificados pelo usuário. Decidiu-se estruturar os arquivos de *log* em campos separados por "," (formato CSV), com a finalidade de tornar mais fácil uma futura análise automatizada do arquivo. Esta convenção também torna possível a visualização e manipulação das informações em planilhas eletrônicas. Os campos de informações definidos para o arquivo são:

1. **Data**: no formato AAAA-MM-DD hh:mm:ss;
2. **Tipo de mensagem**: *string* descrevendo o tipo de mensagem (erro, notícia, etc);
3. **Mensagem**: texto da mensagem. Este campo deve seguir as especificações do formato CSV quanto a existência de caracteres considerados especiais em seu conteúdo, como a ",", por exemplo.

Este gerenciador de *log* não pode ser executado em paralelo, pois isso pode causar uma desordenação na escrita das mensagens para o arquivo. Como não é previsto pelo módulo de

Figura 4.7 – Condição de corrida do módulo executor de ações



Fonte: do autor

integração tratar especificidades de gerenciadores de *log*, o próprio gerenciador implementado realiza a exclusão mútua, utilizando *mutex*.

A Figura 4.8 contém o código do gerenciador de *log* implementado. Este gerenciador abre o arquivo de *log*, processa a mensagem recebida, tratando caracteres especiais do formato CSV, bloqueia um *mutex* e então escreve a mensagem para o arquivo. Após, libera o *mutex* e fecha

o arquivo. A Figura 4.9 contém um trecho de um arquivo de *log* gerado pelo gerenciador implementado.

Figura 4.8 – Código do gerenciador de *log* que escreve em arquivos

```
bool able_logFile(const char* sMessage, const char* sType, const char* sFile) {

  FILE* rsFile;
  char* sTmp;
  time_t oNow;

  rsFile = fopen(sFile, "ab");
  if (rsFile == NULL) {
    return false;
  }
  sTmp = able_strReplace("\"", "\\\"", (char*)sMessage, 0, false);
  sTmp = able_strReplace("\r\n", " ", sTmp, 0, true);
  sTmp = able_strReplace("\r", " ", sTmp, 0, true);
  sTmp = able_strReplace("\n", " ", sTmp, 0, true);
  sTmp = able_strReplace(",", "\\,", sTmp, 0, true);

  /* MUTEX */
  pthread_mutex_lock(&oMutexLog);
  oNow = time(NULL);
  fprintf(rsFile, "%s,%s,\"%s\"\n", able_getFormatedDate(&oNow, "Y-m-d h:i:s"), sType, sTmp);
  pthread_mutex_unlock(&oMutexLog);

  fclose(rsFile);
  free(sTmp);
  return true;

}
```

Fonte: do autor

Figura 4.9 – Trecho de um arquivo de *log* gerado pelo gerenciador implementado

```
1: 2011-12-28 04:26:16,WARNING,"In service 'dns'\, server '10.10.10.2' (ID 2): Server is now unavailable.
   Checking output: Child process timeout."
2: 2011-12-28 04:26:17,NOTICE,"In service 'dns'\, server '10.10.10.1' (ID 1): Redirection is active to the
   BROKEN server '10.10.10.2' (ID 2). Running undo action commands (removing redirection)."
3: 2011-12-28 04:26:18,NOTICE,"In service 'dns'\, server '10.10.10.1' (ID 1): Choosen a substitute server:
   '10.10.10.3' (ID 3). Running action commands (activating redirection)."
4: 2011-12-28 04:26:19,NOTICE,"In service 'dns'\, server '10.10.10.2' (ID 2): Choosen a substitute server:
  '10.10.10.3' (ID 3). Running action commands (activating redirection)."
```

Fonte: do autor

4.2.5 Envio de Notificações

O sistema possibilita o envio de notificações contendo as mensagens geradas internamente. As notificações somente são enviadas quando houver alteração na disponibilidade dos serviços, pois esta é a situação de maior relevância para a notificação dos administradores da rede. Para o envio de notificações, nesta primeira versão do sistema, foi adaptado o cliente para envio de emails MSMTP para ser utilizado como notificador. Porém, ele não comporta-se como um *plugin*, e sim como parte integrante do sistema, com os seus parâmetros definidos individualmente no arquivo de configuração, o que permite maior granularidade na configuração de notificações. Por exemplo, é possível utilizar vários parâmetros globais e somente alterar os

destinatários dos emails para cada servidor sendo monitorado. Este cliente suporta diversos métodos de autenticação e utilização de criptografia TLS.

É possível a utilização de programas externos para notificação. *Plugins* de notificação serão implementados para as próximas versões dos *AvailableD* e serão colocados em */usr/lib/availabled/plugins/alert/*. Para esta primeira versão foi desenvolvido apenas um *plugin* para testes, que escreve as mensagens de notificação em um arquivo de texto, ou seja, funciona de forma semelhante ao gerenciador de *log* implementado.

Na Figura 4.10 está um trecho de código da função responsável por realizar as notificações. Esta função verifica, por primeiro, se há configuração para enviar notificação através do componente adaptado MSMTP. Após, percorre a lista de comandos de notificação, interpreta a variável $MESSAGE e executa os comandos. Os plugins necessitam apenas informar o valor que será atribuído a esta variável para que as notificações sejam encaminhadas aos administradores dos sistemas.

Figura 4.10 – Trecho da função responsável por realizar as notificações

```
/* Notification by email with internal component (MSMTP) */
if (opServiceInfo->bMailNotification) {

  able_setMailData(sMessage, opServiceInfo->opMailConf);
  able_sendMail(opServiceInfo->opMailConf, sError);

} /* End if mail notification */

for (opList = opServiceInfo->opNotificationCommands;
     able_listIsElement(opList);
     opList = opList->opNext) {

  /* Process variables */
  sCmd = (void*)able_strReplace("$MESSAGE", sMessage, (char*)opList->opData, 0, true);

  /* Run notification command */
  iTimeout        = 0;
  rsCommandOutput = able_pipeOpen(sCmd, ABLEFILE_OUTS, sError, -1, -1, &iExitStatus, &iTimeout);
```

Fonte: do autor

4.2.6 Integração

Este módulo cria os *threads* de monitoramento e realiza a integração entre os demais módulos. Os módulos são todos separados em bibliotecas de funções, as quais são responsáveis pelas funcionalidades de cada módulo. O que o módulo integrador faz é simplesmente chamar estas funções, receber as respostas, processá-las ou repassá-las a outros módulos.

Na Figura 4.11 está um trecho da função que cria os *threads* de monitoramento. O que este código faz é percorrer a lista de serviços a serem monitorados, gerada pelo módulo configuração, e obter, para cada serviço, a lista de servidores a serem monitorados. Esta lista é percorrida e,

para cada servidor a ser monitorado, é criado um *thread*.

Figura 4.11 – Código que cria os *threads* de monitoramento

```
/* Start monitoring threads */
for (;able_listIsElement(opServiceList);
     opServiceList = opServiceList->opNext) {

  opService = (able_service_t*)opServiceList->opData;
  for (opServiceInfoList = (able_list_t*)opService->opInfo;
       able_listIsElement(opServiceInfoList);
       opServiceInfoList = opServiceInfoList->opNext) {

    opServiceInfo = (able_svcinfo_t*)opServiceInfoList->opData;

    create_monitoring_thread:
    iThreadRet = pthread_create(&opServiceInfo->oStatus.oThread,
                                &oThreadAttributes,
                                (void*)(&able_serverMonitoring),
                                (void*)opServiceInfo
                               );
```

Fonte: do autor

A Figura 4.12 contém um trecho de código dos *threads* de monitoramento. Neste trecho de código, está o tratamento para dois resultados de verificação de disponibilidade: erro na verificação e serviço indisponível. A verificação do serviço é realizada através da função *able_checkerRun* do módulo monitoramento. Esta função recebe as informações da configuração (*opServiceInfo*), que contém o *plugin* de verificação a ser utilizado, e retorna o resultado da verificação (*iCheckRet*) e as mensagens enviadas para a saída padrão do *plugin* (*sOutput*). Caso a verificação não tenha sido realizada devido a algum erro (ABLECHECK_ERROR), uma mensagem é gerada e enviada ao módulo log através da função *able_log*, que recebe a mensagem de *log* e as informações de configuração sobre os gerenciadores de *log*. Quando é detectada a indisponibilidade do serviço (ABLECHECK_UNAVAILABLE), os redirecionamentos de outros servidores para o servidor indisponível são removidos e novos servidores substitutos são escolhidos, o que é realizado pela função *able_manageRedirectionsToBrokenServer* do módulo executor de ações. Após, a função *able_executorAction* realiza o redirecionamento do servidor indisponível para uma réplica disponível. Por fim, é realizado *log* e o módulo notificação é invocado através da função *able_notifierRun*.

4.2.7 Instalação

Para auxiliar a instalação, no que diz respeito a configurações necessárias do sistema alvo de instalação, foi utilizado o *autotools*. Esta é uma ferramenta comumente utilizada por programas escritos em C, pois automatiza todo o processo de instalação, tornando-a fácil. Quando utilizada esta ferramenta, para proceder com a instalação de sistemas, geralmente são necessários apenas três passos:

Figura 4.12 – Trecho de código dos *threads* de monitoramento

```
iCheckRet = able_checkerRun(opServiceInfo, sOutput);
switch (iCheckRet) {

  case ABLECHECK_ERROR:

    /* Set log message */
    able_setLogMsg(ABLELOG_ERROR, opLogMsg,
                   ABLEMSG_IN_SERVICE_SERVER" "
                   ABLEMSG_CHECK_ERROR,
                   ((able_service_t*)opServiceInfo->opService)->sName,
                   opServiceInfo->sIP, opServiceInfo->iID, sOutput[0]
                  );
    able_log(opLogMsg, opServiceInfo->opLogInfo);

  case ABLECHECK_UNAVAILABLE:

    /* => LOCK */
    pthread_mutex_lock(&oMutexStatus);
    /* If the service was available before, action commands should be executed. */
    if (opServiceInfo->oStatus.bAvailable) {

      /* Set service status to unavailable */
      opServiceInfo->oStatus.bAvailable = false;


      able_manageRedirectionsToBrokenServer(opServiceInfo);

      /* Try redirect to a substitute server */
      able_executorAction(opServiceInfo);

      /* Set log message */
      able_setLogMsg(ABLELOG_WARNING, opLogMsg,
                     ABLEMSG_IN_SERVICE_SERVER" "
                     ABLEMSG_SERVER_IS_NOW_UNAVAILABLE,
                     ((able_service_t*)opServiceInfo->opService)->sName,
                     opServiceInfo->sIP, opServiceInfo->iID, sOutput[0]
                    );
      able_log(opLogMsg, opServiceInfo->opLogInfo);

      /* Notify */
      able_notifierRun(opLogMsg->sMessage, opServiceInfo);

    } /* End if (opServiceInfo->oStatus.bAvailable) */
    /* => UNLOCK */
    pthread_mutex_unlock(&oMutexStatus);
```

Fonte: do autor

1. Executar o *script* de configuração gerado pela ferramenta (*configure*). Este *script* analisa o sistema em busca das bibliotecas necessárias, programas auxiliares, diretórios para instalação, dentre várias outras coisas. Se o programa sendo instalado possui configurações que podem ser definidas pelo usuário, basta passar seus valores para o *script* que ele preparará o *software* para ser instalado com valores definidos pelo usuário. Para exemplificar, considere uma aplicação onde o arquivo de *log* é especificado no momento de instalação. Um exemplo de passagem do nome do arquivo poderia ser:

   ./configure --log-file=/var/log/app.log;

2. Executar o comando *make*: este comando irá compilar e construir o *software*, de acordo com sua configuração;

3. Executar o comando *make install*: irá copiar os arquivos do *software* para seus respectivos lugares, inclusive manuais e documentação, quando houverem.

É comum a utilização do *autotools* em ferramentas desenvolvidas para ambientes GNU/-Linux, incluindo serviços de rede como o Apache2 (servidor HTTP) e o Bind (servidor DNS). Portanto, é comum administradores de rede já terem instalado sistemas que utilizam o *autotools* e já conhecerem os passos a serem executados.

O *AvailableD* possui apenas dois parâmetros que podem ser configurados pelo usuário. Estes parâmetros dizem respeito a configurações do MSMTP (utilizado para enviar emails) e são:

1. *--with-ssl*: determina se vai ser compilado com suporte a TLS. Se sim, deve ser informada qual biblioteca será utilizada. O suporte a TLS, geralmente não é importante quando forem utilizados servidores de email da rede interna, pois os riscos de problemas de segurança são baixos. Quando utilizados servidores externos, é interessante usar criptografia. Valores aceitos: *gnutls*, *openssl* e *no*;

2. *--with-libgsasl*: determina se será utilizada a biblioteca GNU SASL. Esta é uma biblioteca que possibilita que vários métodos de autenticação sejam utilizados com os servidores de email. Este parâmetro não aceita valores, simplesmente informá-lo determina que a biblioteca deverá ser utilizada.

Futuramente, pretende-se gerar pacotes de instalação que podem ser utilizados por várias distros GNU/Linux. Como exemplo de pacotes bastante utilizados, pode-se citar pacotes RPM e DEB. A vantagem dos pacotes é a maior facilidade de instalação que podem proporcionar. Tomando como exemplo pacotes DEB, a instalação pode ser realizada, inclusive, através de clique duplo no arquivo do pacote.

4.2.8 Operação

Para operacionalizar o AvailableD após sua instalação é necessário configurar os parâmetros de operação do sistema através do arquivo de configuração. É possível verificar a configuração e identificar problemas sem iniciar o *AvailableD*, e verificar alterações realizadas enquanto o mesmo já está rodando, sem que sejam efetivadas. Para isso, basta passar o argumento *--check-conf* para o executável do *daemon*. É possível também informar o arquivo de configuração a ser verificado, passando o argumento *--config-file* seguido do caminho para o arquivo. Ex:

/usr/sbin/availabled --check-conf --config-file /user/home/conf.bkp

Depois de configurado, basta iniciar o *daemon*, passando o parâmetro *start* para o executável:

/usr/sbin/availabled start

Erros na inicialização do *daemon* ou problemas detectados na configuração serão informados ao usuário através de mensagens de erro detalhadas, por vezes seguidas de dicas para resolução dos problemas identificados, de forma a facilitar e agilizar a resolução dos problemas. Quando o *daemon* for inicializado com sucesso, é criado um arquivo com o PID (código do processo) do *AvailableD* em /var/run/availabled.pid. Este arquivo é utilizado para parar o *daemon*.

A parada do *daemon* é realizada passando-se o parâmetro *stop* para o executável. Na ocasião de alteração da configuração, é necessário parar e depois iniciar novamente o *AvailableD*. Para agilizar este processo, foi implementado o parâmetro *restart*, que executa as duas operações em sequência. Pretende-se implementar a possibilidade de recarregar a configuração sem necessitar parar o *daemon*.

5 Testes e Resultados

A fase de testes e validação é uma parte importante do trabalho, pois visa demonstrar, com dados e estatísticas, que os objetivos postos foram atingidos. Para este fim, foram realizados testes funcionais a fim de detectar erros e validar o protótipo implementado de acordo com suas especificações. Para mensurar a eficiência da solução, foram executados testes de desempenho, onde foram avaliados a utilização de recursos pela execução da solução e o potencial aumento da disponibilidade gerado pela utilização do AvailableD.

5.1 Ambiente de Testes

Para verificação do funcionamento do sistema, foi preparado um ambiente de testes composto por máquinas virtuais, com o auxílio da ferramenta de virtualização VirtualBox (`https://www.virtualbox.org/`). O *firewall*, onde foi instalado o AvailableD, foi o único não virtualizado, cosistindo de uma máquina física com sistema operacional Ubuntu 9.10, que vem com o *firewall* IPTables integrado. A configuração física desta máquina consiste em: a) Processador Pentium(R) dual-core de 2,2 GHz; b) 4 GBs de memória principal; c) 320 GBs de espaço em disco e d) 1 placa de rede 10/100Mbps, conectada a uma porta de 100Mps de um *switch* 10/100Mbps. Na mesma máquina do *firewall*, foi virtualizado um cliente dos serviços com sistema Ubuntu 10.04, o qual foi colocado na subrede 172.16.0.0/16, comunicando-se com o *firewall* por meio de uma interface de *bridge* (interface virtual que liga duas ou mais redes)com o IP 172.16.0.1.

Para a realização dos testes, foram utilizados dez servidores (réplicas), com o sistema Debian 6.0.2. Cada réplica contém os seguintes serviços em execução (replicados):

1. **DNS**: utilizado para tradução de nomes em endereços IP. O *software* utilizado foi o Bind na versão 9.7.3;

2. **SMTP**: serviço para envio de emails. O servidor de email instalado foi o Postfix na versão 2.7.1;

3. **HTTP**: serviço para transferência de hipertexto. Utilizou-se como servidor HTTP o Apache2 versão 2.2.16.

Os servidores consistiram em máquinas virtuais (MVs), que foram colocadas em máquinas físicas, sendo um servidor por máquina física, a fim de tornar mais semelhante a um servidor

físico. A configuração física destas máquinas consiste em: a) Processador Intel(R) Core(TM) i5 de 3.2 GHz; b) 3,6 GBs de memória principal; c) 300 GBs de espaço em disco e d) 1 placa de rede 10/100Mbps, conectada a uma porta de 100Mps de um *switch* 10/100Mbps. A opção por virtualizar os servidores em vez de utilizar os próprios sistemas das máquinas físicas, deu-se por facilitar e agilizar a criação do ambiente de testes, já que suprime a necessidade de instalação e configuração dos serviços nos sistemas hospedeiros, instalados nas máquinas físicas.

Os servidores, tanto físicos como virtuais, foram colocados na subrede 10.10.10.0/24. Os endereços IP para as máquinas virtuais atribuídos, ficaram na faixa 10.10.10.1 a 10.10.10.10, configurados estaticamente. A comunicação com o sistema hospedeiro foi realizada através de *bridging* com a interface de rede física da máquina. Para as interfaces físicas, foram atribuídos, de forma estática, os endereços IP de 10.10.10.101 a 10.10.10.110.

A comunicação entre o *firewall* e as demais máquinas (servidores) deu-se através da placa física, que recebeu o endereço IP 10.10.10.254. Para possibilitar a comunicação entre o cliente com os servidores, separados em redes distintas, foi definido como roteador de ambos o próprio *firewall*, que possui interfaces nas duas redes. As máquinas foram conectadas através de cabos UTP e *switches Ethernet* de 100Mbps.

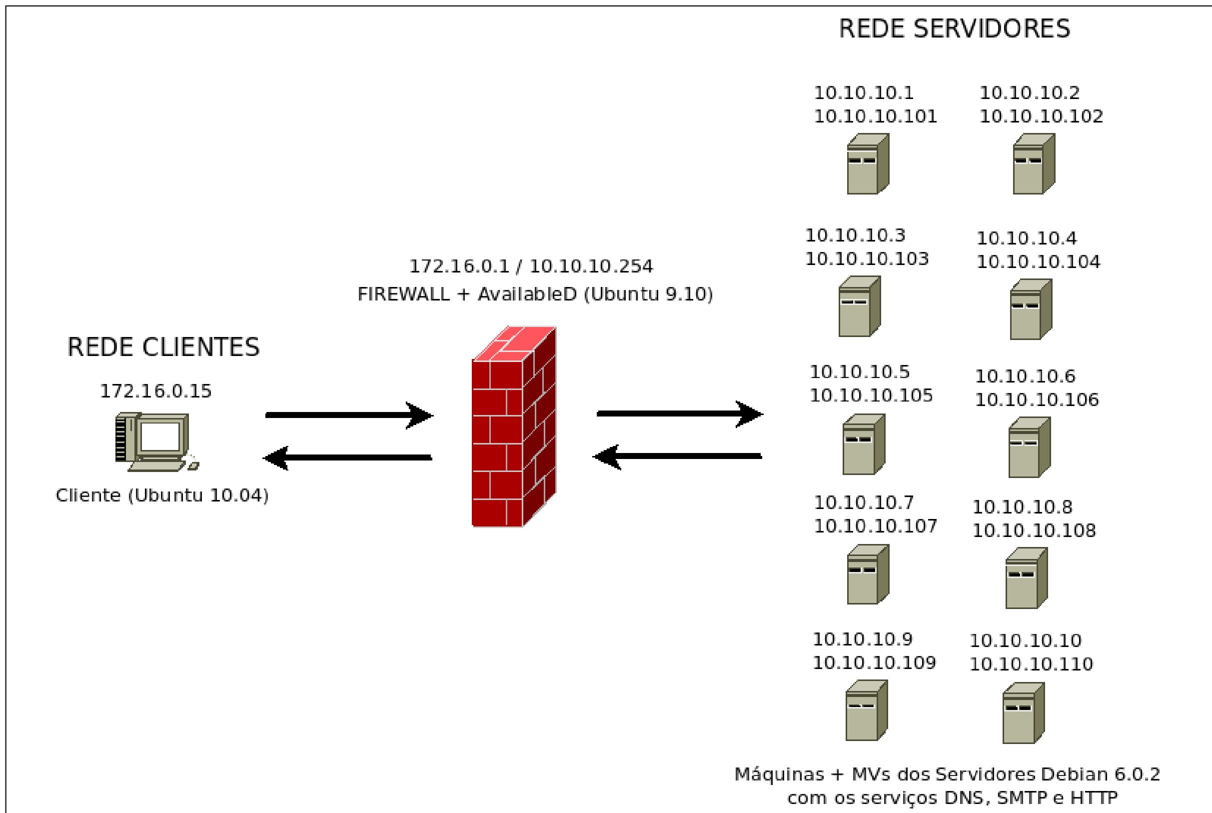
Na Figura 5.1 está representado graficamente o ambiente de testes. A rede dos servidores é composta por dez máquinas físicas, cada uma contendo uma MV. A rede dos clientes contém apenas uma máquina virtual, que realiza o acesso aos serviços providos pelas diferentes réplicas.

5.2 Testes Funcionais

Com o objetivo de identificar erros e validar o funcionamento do protótipo implementado de acordo com suas especificações, foram realizados testes funcionais. As funcionalidades do AvailableD, primeiramente, foram testadas de forma isolada. Somente após cada funcionalidade ser validada e erros não serem mais encontrados, os módulos do sistema foram totalmente integrados e testados em cojunto.

A técnica utilizada nos testes funcionais consistiu na utilização de diferentes entradas para cada funcionalidade. Várias configurações foram testadas para o AvailableD. Procurou-se testar a maior quantidade de condições de erros possíveis, utilizando-se entradas de forma a forçar ocorrências de falhas no sistema. Por exemplo, para testar o gerenciador de *log* implementado, forçando condições de erro, foram informados caminhos para arquivos de *log* inexistentes, arquivos sem permissão de escrita, as permissões foram alteradas e os arquivos foram excluídos durante a execução do sistema. Os testes realizados permitiram a identificação e correção de

Figura 5.1 – Ambiente utilizado para testes do *AvailableD*



Fonte: do autor

diversos problemas. As funcionalidades testadas e validadas estão descritas a seguir:

1. **Instalação**: foi testada a geração dos arquivos e *scripts* de instalação pelo autotools. Os parâmetros de instalação informados pelo usuário foram testados com vários valores diferentes.

2. **Carregamento da configuração**: testou-se o carregamento de diversas configurações, incluindo configurações com erro de sintaxe, inválidas e incompletas. O valor de cada parâmetro de configuração foi verificado em cada teste realizado;

3. **Verificação de disponibilidade dos serviços**: foram testados os três *plugins* verificadores implementados e a comunicação dos mesmos com o AvailableD. Foram simuladas condições de falhas nos serviços, utilizando-se um *shell script* desenvolvido com este propósito. Este *script* realiza a parada e inicialização dos serviços em intervalos de tempo configuráveis;

4. **Alteração das regras de roteamento**: com o auxílio do *script* para simulação de falhas nos serviços, foram testadas diversas adições e remoções de redirecionamentos para

as réplicas dos serviços. A verificação da corretude dos redirecionamentos adicionados/removidos, foi realizada através da consonância entre a tabela de regras de roteamento ativas do IPTables, e as réplicas ativas e inativas dos serviços;

5. **Envio de notificações**: o envio de notificações foi testado tanto pela utilização do MSMTP, quanto de *plugins* notificadores. O único *plugin* notificador testado foi o *plugin* para notificação através de arquivos, desenvolvido unicamente com o intuito de testes;

6. ***Log* de informações**: foi testado o gerenciador para *log* em arquivos. Várias mensagens de *log*, definidas estaticamente, foram transmitidas ao gerencidador, para que realizasse o armazenamento das mesmas nos arquivos de *log* em formato CSV.

7. **Iniciar, parar e reiniciar o *daemon***: Os três comandos de operação do *daemon* implementados foram testados após a integração dos módulos.

5.3 Testes de Desempenho

Os testes de desempenho do AvailableD foram realizados após o término dos testes funcionais. O objetivo dos testes de desempenho foi permitir a análise estatística de características de desempenho relevantes do AvailableD. As características analisadas foram: utilização da unidade central de processamento (UCP), quantidade de memória virtual alocada, tráfego de rede gerado pelas verificações dos serviços e tempo em resposta a falhas.

Para a execução dos testes de desempenho, foram utilizadas, no total, vinte configurações diferentes para o *AvailableD*, sendo utilizadas nove configurações para obter o tempo em resposta a falhas, e todas as vinte configurações para coletar os demais dados. A diferença entre as configurações consistiu apenas no intervalo de tempo entre as verificações de disponibilidade realizadas em cada serviço para cada servidor, e no número de monitoramentos. Foram utilizados quatro intervalos de tempo diferentes: cinco, dez, quinze e vinte segundos. Para cada intervalo de tempo, foram configurados cinco diferentes números de monitoramentos, determinados pelo número de réplicas monitoradas (cada uma com três serviços): duas réplicas (seis monitoramentos), quatro réplicas (doze monitoramentos), seis réplicas (dezoito monitoramentos), oito réplicas (vinte e quatro monitoramentos) e dez réplicas (trinta monitoramentos), O objetivo destas diferentes configurações consiste em identificar o impacto do intervalo de tempo entre cada verificação e do número de monitoramentos realizados sobre o tempo médio para identificação de falhas e adição dos redirecionamentos, e sobre a utilização de UCP, memória virtual e tráfego de rede gerado.

A verificação da disponibilidade dos serviços foi realizada com os *plugins* implementados. Para o serviço HTTP, a verificação era feita pela requisição de uma página HTML, utilizando o *plugin* http.monitor. O serviço SMTP era verificado pelo *plugin* smtp.monitor, que executa alguns comandos SMTP. Para verificar o serviço DNS, utilizou-se o *plugin* dns-query.monitor, realizando uma consulta DNS de um nome de domínio configurado no servidor.

O ideal é que o tempo de verificação de cada serviço seja o menor possível, diminuindo assim a percepção das réplicas falhas por parte dos clientes. Os principais fatores que podem deixar o monitoramento mais lento são: a) excesso de carga no servidor das réplicas; b) tráfego de rede saturado; c) rede problemática; d) excesso de serviços a serem monitorados; e) tempo de detecção de um serviço falho, incluindo *timeout* de protocolos, tempo de resposta, entre outras variáveis. O último fator é especialmente crítico, pois ele não permite que o tempo de verificação tenda ou seja zero. Logo, o desafio é identificar o menor tempo possível sem levar a falsos positivos.

Para definir os tempos máximos de cada verificação para cada serviço, utilizou-se uma funcionalidade implementada no *AvailableD* com este propósito. Esta funcionalidade consiste em parâmetros que são passados ao *AvailableD* informando os serviços que serão testados e o número de testes a serem realizados. Após realizar os testes, são exibidas as estatísticas dos resultados das verificações para cada serviço/servidor, exibindo o número de verificações em que foi constatada disponibilidade, indisponibilidade e o número de verificações que não puderam ser realizadas, assim como as últimas mensagens de erro recebidas, o que ajuda identificar a incorretude nos dados de verificação configurados e também os motivos de indisponibilidade, como, por exemplo, o estouro do *timeout* de verificação. Os parâmetros necessários informar para utilizar esta funcionalidade são:

1. --config-file *arquivo*: indica o caminho para o arquivo de configuração, com as informações dos monitoramentos, a ser utilizado. Se não informado, o arquivo de configuração padrão (/etc/availabled.conf) é utilizado;

2. --test-repeat *número*: informa o número de vezes que os testes deverão ser realizados. Se não informado, assume o valor um;

3. --test-check *monitoramentos*: lista os monitoramentos configurados que deverão ser testados. O formato da lista informada é: nome_serviço1:id_servidor1,id_servidor2...%nome_serviço2....

Os servidores do ambiente de testes possuem configurações iguais. A carga dos servidores é baixa, consistindo apenas das requisições realizadas durante o monitoramento e o tempo de

comunicação entre as máquinas é pequeno e muito semelhante entre os pares de máquinas, pois trata-se de uma rede local estável e estruturada, onde todos os componentes são interconectados através de comutadores. Sendo assim, o tempo máximo de verificação pode ser configurado com valores iguais para todos os servidores do mesmo serviço. Após teste de vários valores, foi possível identificar quais os menores valores aceitáveis, isto é, com baixos riscos de ocasionar identificação errônea de indisponibilidade. Dentre os valores aceitáveis, selecionou-se o tempo de dois segundos como tempo limite de cada verificação, para os três serviços.

Os *plugins* de redirecionamento utilizados nos testes foram comandos do IPTables que inserem regras na tabela de roteamento NAT. Estas regras, são responsáveis por alterar os endereços e portas de destino, quando os mesmos pertencerem aos servidores nos quais foi diagnosticado a indisponibilidade do serviço. As regras também levam em consideração o protocolo de transporte utilizado, sendo TCP para os serviços HTTP e SMTP e UDP para o serviço DNS.

Para *log* de informações, foi utilizado o gerenciador de *log* que escreve em arquivos, com um único arquivo para todos os serviços. O envio de notificações foi desabilitado durante os testes. Não há a necessidade de emitir notificações, pois somente durante os testes para verificação do tempo em resposta a falhas, de fato, seriam enviadas notificações (devido a simulação de falhas). Porém, o envio de notificação não afeta o tempo em resposta a falhas, já que é realizado após a manipulação de regras no *firewall*.

O conteúdo do arquivo de configuração utilizado para monitorar duas réplicas (seis monitoramentos), com intervalo de cinco segundos entre as verificações está na figura I.1 do Apêndice I. As demais configurações utilizadas mudam apenas no parâmetro *wait_time* (tempo entre as verificações), e nas definições *server*, que determinam os servidores monitorados.

Os aspectos analisados na máquina onde o *AvailableD* é executado (*firewall*) foram a utilização da UCP, a memória virtual alocada e o tráfego de rede gerado. As informações a respeito destes aspectos foram coletadas com auxílio de ferramentas com este propósito. Para capturar as informações de utilização da UCP e quantidade de memória alocada foi utilizado o utilitário Top do pacote de ferramentas Procps `http://procps.sourceforge.net/`. O tráfego de rede foi capturado com o auxílio do *script* Bytetraf `http://comp.eonworks.com/scripts/monitor_network_traffic-20040720.html`. Para obter, processar e organizar as informações, foram desenvolvidos três *scripts*, sendo um para cada tipo de informação. Estes *scripts* capturam as informações a cada período de tempo configurável, processam, e exibem na saída padrão, sendo impressa uma informação coletada em cada linha. Os *scripts* desenvolvidos para obtenção das informações de utilização da UCP, memória virtual alocada e tráfego de rede

encontram-se nas Figuras H.1, H.2 e H.3 do Apêndice H, respectivamente.

A coleta das informações de utilização da UCP, quantidade de memória virtual alocada e tráfego de rede gerado, foi realizada para cada uma das vinte configurações. Para cada configuração, os dados foram coletados cinco vezes, cada uma com tempo de coleta de cinco minutos e intervalo entre cada obtenção de dados de um segundo, o que dá um total de trezentos dados obtidos em cada coleta. Antes de cada coleta, o AvailableD era reconfigurado (quando necessário) e reiniciado.

Para obter a duração dos *outages* (ou *donwtimes*) (tempo de indisponibilidade) percebidos pelos clientes dos serviços, o que permite o cálculo do tempo médio em resposta a falhas do AvailableD, foram utilizadas nove configurações, que são a combinação dos tempos de cinco, dez e quinze segundos entre os monitoramentos, e os números de seis, doze e dezoito monitoramentos realizados. Para coletar os *outages*, faz-se necessário simular falhas nos serviços que estão sendo acessados pelo cliente. Com o objetivo de simular as falhas, foi desenvolvido um *script*. A sua função é, periodicamente, parar (se estiverem executando) ou iniciar (se estiverem parados) os serviços na réplica principal (réplica que o cliente conhece o endereço), que, para os testes, ficou definida a réplica de IP 10.10.10.1. Com este *script*, os serviços ficavam indisponíveis por períodos de tempo, o que permitia a verificação da continuidade dos mesmos por parte dos clientes.

O *script* para simulação de falhas foi desenvolvido em *shell script*. Seu código está na figura H.4 do Apêndice H. Este *script* simplesmente para/inicia os serviços após um tempo fixo em segundos, configurável. Além de realizar a simulação de falhas, o *script* registra o *downtime* real aproximado de cada serviço, o que é interessante de comparar com os valores de *downtime* percebidos pelos clientes. Estes valores são exibidos quando o *script* é finalizado através do envio dos sinais 1 (SIGHUP) ou 2 (SIGINT), sendo o último emitido através da combinação das teclas *ctrl + c*. O tempo total de execução do *script* também é mostrado.

Para acessar os serviços a partir da estação cliente, foram desenvolvidos *scripts*. Estes *scripts* foram responsáveis por realizar verificações de disponibilidade e coletar os dados necessários para mensurar os níveis de disponibilidade de cada serviço. As verificações realizadas, no caso dos serviços HTTP e SMTP, consistiram em simplesmente abrir um *socket* na porta de escuta do serviço. Sempre que não fosse possível abrir o *socket*, dava-se início a contagem do *downtime*. Para testar estes serviços, foram necessários dois *scripts*, um desenvolvido em *PERL*, para abrir a conexão, e outro desenvolvido em *shell script*, responsável por executar o *script* PERL e determinar o tempo de duração de cada período de indisponibilidade, tempo este que demonstra o efeito da utilização do *AvailableD*, pois quando é realizado o redirecionamento,

o serviço volta a estar disponível aos clientes, o que reduz o *downtime* percebido pelos clientes.

Na figura H.5 do Apêndice H está o código do *script* PERL responsável por abrir o *socket*. Este *script* considera um *timeout*, o que evita grandes tempos de espera para abertura do *socket*. A maneira de identificar se foi ou não possível abrir o *socket* é através do código de saída, sendo 0 para quando o *socket* foi aberto e 1 para quando não foi.

O código do *shell script* que identifica os períodos de indisponibilidade para os serviços HTTTP e SMTP encontra-se na figura H.6 do Apêndice H. Este *script* recebe como parâmetros o endereço do servidor, a porta de escuta e um *timeout* para abertura do *socket*, que são passados ao *script* PERL descrito anteriormente. Quando for parado através dos sinais SIGHUP ou SIGINT, emite o tempo total de execução e o tempo total de *downtime*. O tempo de duração de cada período de indisponibilidade identificado é emitido sempre que cada *outage* chega ao fim.

Para verificar a disponibilidade do serviço DNS e coletar os tempos de *downtime* a partir da máquina cliente, foi desenvolvido um *shell script* que utiliza a ferramenta NSlookup `http://enc.com.au/itools/nslookup.php` para realizar as consultas DNS. Quando a consulta falhava iniciava-se a contagem da duração de cada período de indisponibilidade. Os dados coletados e emitidos são os mesmos dos *scripts* desenvolvidos para verificar os serviços HTTP e SMTP. O código deste *script* encontra-se na figura H.7 do Apêndice H.

Para os *scripts* de coleta de dados da disponibilidade dos serviços, o tempo médio entre cada coleta utilizado foi de apenas um segundo. Este curto intervalo de tempo permite identificar com maior precisão a duração dos períodos de indisponibilidade e, mostrou-se suficiente para que não fossem gerados falsos positivos de indisponibilidade. Os dados emitidos pelos *scripts* são enviados para a saída padrão, sendo uma informação em cada linha, o que facilita o processamento automatizado dos dados coletados.

Durante os testes, as falhas ocorridas (paradas) em cada serviço foram bastante frequentes, pois utilizou-se intervalo de tempo de trinta segundos entre cada parada/inicio dos serviços no *script* para simulação de falhas. Isto gera um tempo de duração de aproximadamente 30 segundos para cada *downtime*/*uptime*, tempo suficiente para o AvailableD realizar até duas verificações dos serviços, já que o intervalo máximo utilizado entre os monitoramento foi de quinze segundos.

A coleta de dados da disponibilidade dos serviços, para cada uma das nove configurações utilizadas, foi realizada pela máquina cliente até atingir informações de sete *outages*. Isso resultou na demora de pouco mais de sete minutos de coleta de dados para cada configuração, pois o *script* que simula falhas nos serviços altera a situação dos serviço a cada trinta segundos, o que resulta em um período completo de *downtime* a cada um minutos (trinta segundos disponível

e trinta segundos indisponível). No cliente, o acesso aos serviços dava-se através do endereço IP de apenas uma das réplicas (10.10.10.1), na qual foi executado o *script* para simulação de falhas.

5.4 Resultados Obtidos

Através dos testes funcionais, foi possível identificar e corrigir vários erros. O funcionamento do protótipo implementado foi validado, pois atendeu as especificações definidas no documento de projeto, Apêndice G, o que permite concluir que a solução desenvolvida atingiu seus objetivos funcionais.

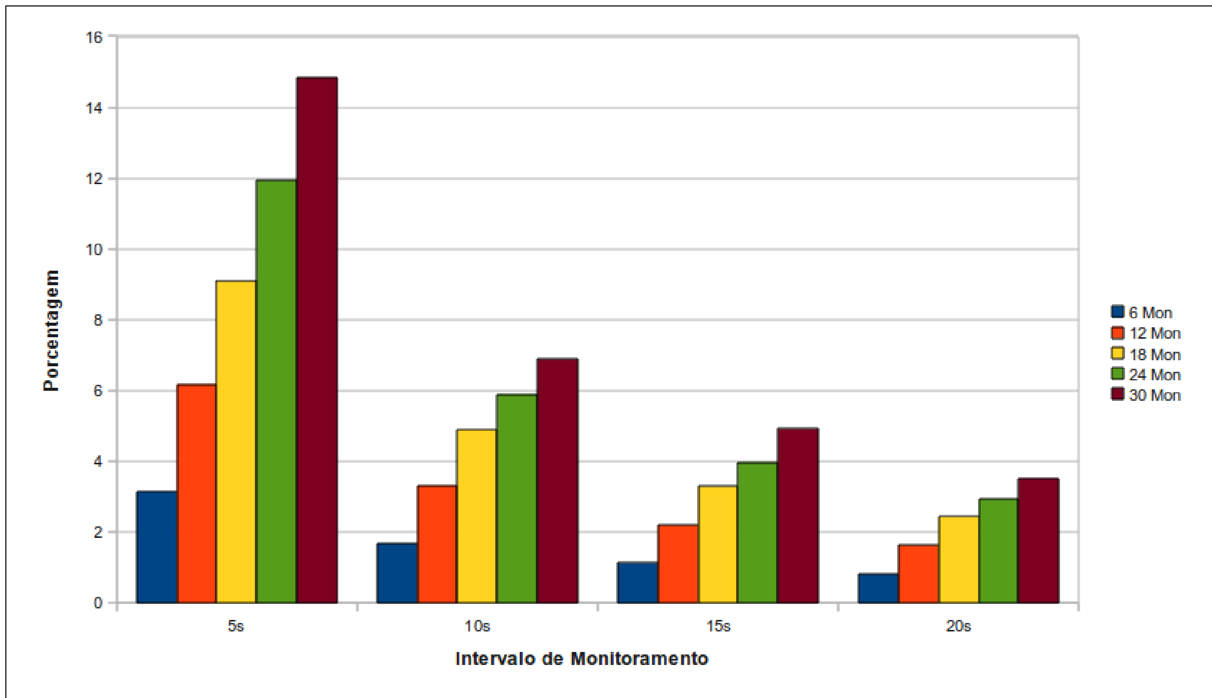
Através dos dados coletados nos testes de desempenho, foram analisadas as métricas de desempenho e identificou-se o nível de eficiência e escalabilidade do protótipo implementado, o que foi possível através da comparação dos resultados obtidos com as diferentes configurações utilizadas pelo AvailableD.

Nas Figuras 5.2, 5.3 e 5.4, estão os gráficos da utilização média de UCP, quantidade de memória virtual alocada e tráfego de rede gerado, respectivamente, para as vinte configurações do AvailableD utilizadas durante os testes.

Através da Figura 5.2, onde está o gráfico da utilização média de UCP, percebe-se que a utilização aumenta conforme aumenta o número de monitoramento ou diminui o intervalo entre as verificações. A utilização mais onerosa, ficou em aproximadamente quinze porcento, onde foram configurados trinta monitoramentos e verificações a cada cinco segundos. Considerando a máquina física utilizada para o *firewall*, cujo processador posui apenas dois núcleos, e visto que há uma notável redução na utilização da UCP com o aumento do tempo entre as verificações, a utilização média de UCP pode ser considerada adequada, pois, mesmo um intervalo de vinte segundos entre as verificações, pode ser considerado um tempo bastante pequeno para a maioria dos serviços prestados em redes locais de pequeno e médio porte.

A quantidade média de memória virtual alocada, que pode ser visualizada na Figura 5.3, variou de acordo com o número de monitoramentos realizados. O tempo entre as verificações de disponibilidade dos serviços praticamente não exerceu influência na quantidade de memória alocada. Isso deve-se ao fato de pequenas quantidades de memória serem necessárias pelos *plugins* verificadores utilizados, que são programas pequenos e simples, com um pequeno período de execução. Observou-se um siginificativo aumento na alocação de memória com o aumento do número de monitoramentos, que ficou em torno de 50 megabytes para cada seis monitoramentos adicionados, isto é, 8,33 megabytes por monitoramento. Este valor está relativamente

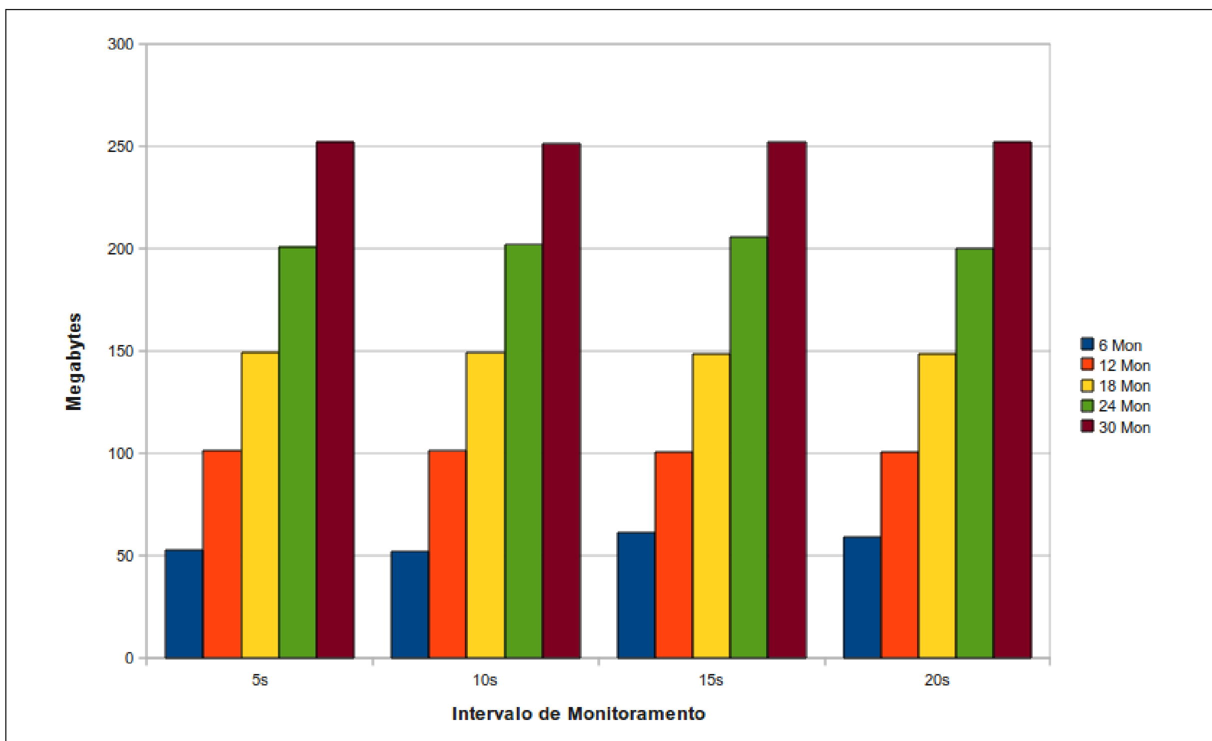
Figura 5.2 – Utilização média da UCP em 300 segundos de monitoramento



Fonte: do autor

elevado, o que afeta a escalabilidade da solução.

Figura 5.3 – Alocação média de memória virtual em 300 segundos de monitoramento

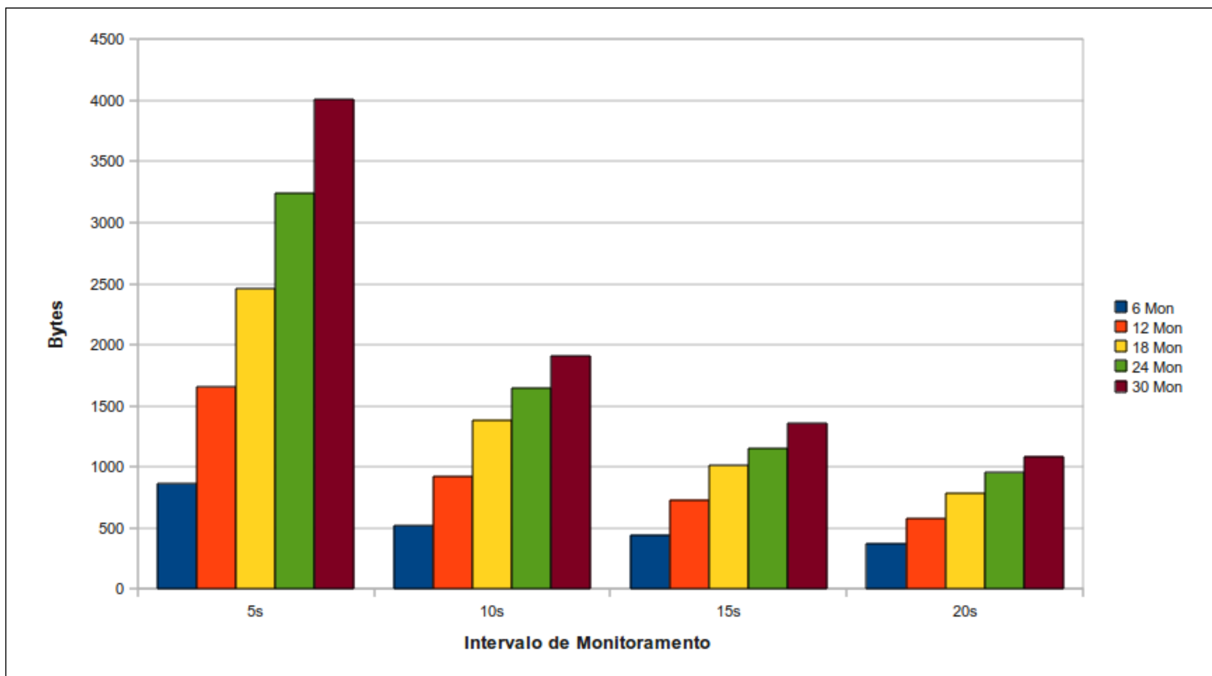


Fonte: do autor

O tráfego de rede médio gerado pelas verificações, que pode ser observado na Figura 5.4,

manteve um comportamento semelhante a utilização de UCP: aumenta conforme diminui o tempo entre cada verificação e conforme aumenta o número de monitoramentos realizados. O aumento no tráfego de rede deu-se na escala de bytes, atingindo o pico médio em torno de quatro mil bytes, quando foram configurados trinta monitoramentos com intervalo de cinco segundos entre as verificações. O aumento observado no tráfego de rede, bem como o pico médio, são valores baixos e adequados, principalmente para redes locais, onde a largura de banda disponível geralmente é elevada, na casa de mega ou gigabits.

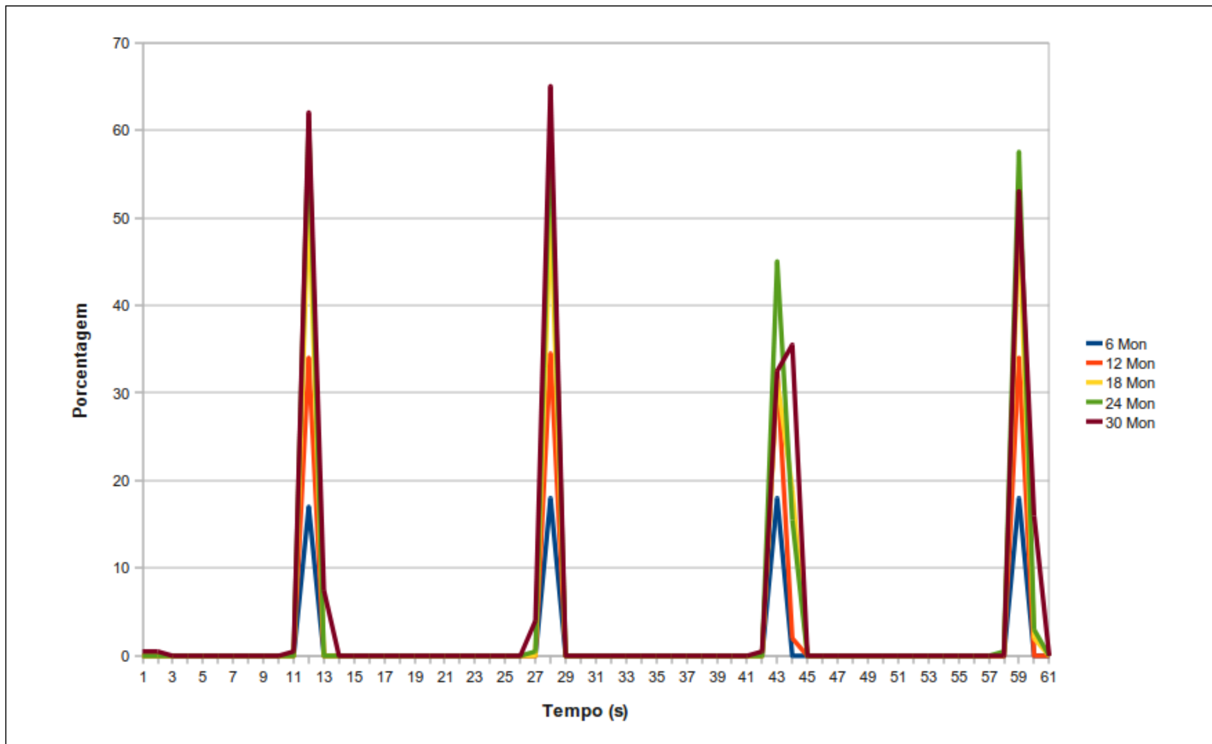
Figura 5.4 – Tráfego de rede médio gerado em 300 segundos de monitoramento



Fonte: do autor

O protótipo implementado cria os *threads* de monitoramento em sequência, dando início às verificações logo após a criação de todos os *threads*. Isso faz com que o início das verificações em cada *thread* seja muito próximo. Como para os testes foram configurados tempos de espera iguais entre as verificações dos serviços, isto teve como consequência a concentração da execução das verificações, causando picos no consumo de alguns recursos, como a utilização de UCP e o tráfego de rede gerado. O gráfico da Figura 5.5 demonstra a ocorrência dos picos na utilização da UCP para o intervalo de quinze segundos entre as verificações. No gráfico está evidente o resultado gerado pela concentração das verificações. Pode-se perceber que a utilização da UCP permanece praticamente nula a maior parte do tempo, atingindo picos com duração de até três segundos. A distribuição das verificações no tempo seria conveniente, pois picos elevados na utilização de recursos, principalmente da UCP, podem gerar períodos de sobrecarga e mau funcionamento.

Figura 5.5 – Utilização da UCP para um intervalo entre verificações de 15 segundos

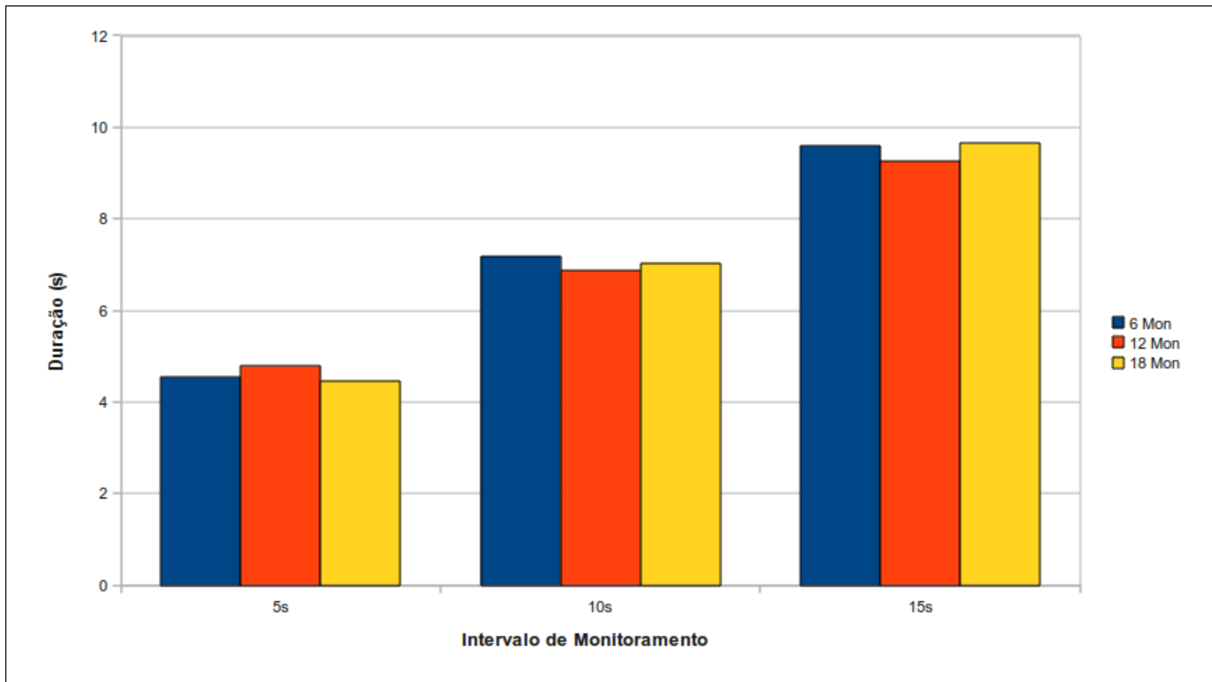


Fonte: do autor

A captura da duração dos períodos de indisponibilidade dos serviços percebidos pelos clientes possibilitou observar o potencial aumento nos índices de disponibilidade. Na figura 5.6 estão as durações médias dos períodos de indisponibilidade percebidos pelos clientes. Os *outages* reais tiveram a duração de aproximadamente trinta segundos, porém, no lado dos clientes, a duração aproximada dos *outages* variou de acordo com o tempo entre as verificações de disponibilidade configuradas no AvailableD. Não constatou-se relação entre o número de monitoramentos e o tempo em resposta a falhas para as configurações utilizadas. A redução no tempo entre as verificações possibilitou a identificação e tratamento das falhas ocorridas nos serviços de forma mais rápida, porém, resulta em maior consumo de alguns recursos. Sendo assim, é necessário conciliar o tempo entre as verificações de cada serviço de forma a obter tempos razoáveis para identificação das falhas sem comprometer a utilização dos recursos.

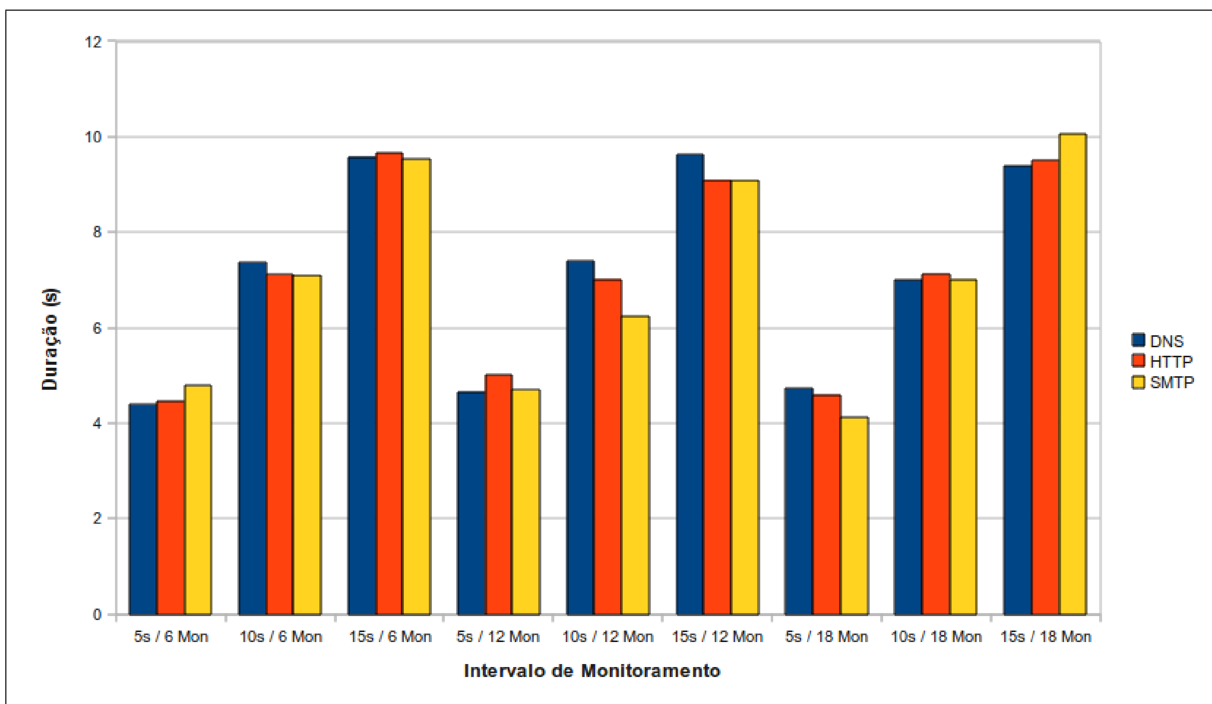
A duração média dos *outages* percebidos pelos clientes também foi analisada entre os três serviços. Através do gráfico da figura 5.7, observa-se que a diferença de resposta a falhas entre os serviços monitorados foi pequena e de pouca significância. Esta diferença é influenciada por muitos fatores como a carga e capacidade da rede e do servidor, características do serviço e a forma como for implementado o *plugin* verificador.

Figura 5.6 – Duração média dos períodos de indisponibilidade percebidos pelos clientes



Fonte: do autor

Figura 5.7 – Comparação dos períodos de indisponibilidade entre os serviços



Fonte: do autor

5.5 Comparação com Outras Soluções

A fim de identificar os pontos positivos e negativos do *AvailableD* e permitir a comparação com as soluções Linux-HA, LVS e Ultra Monkey, faz-se necessário analisá-lo sob os mesmos

aspectos avaliados nestas soluções:

1. **Complexidade de instalação e de configuração**: para o desenvolvimento do *AvailableD* foram utilizados padrões largamente difundidos entre administradores de rede, como configuração realizada por meio de arquivo, a prória sintaxe definida para o arquivo e a instalação realizada com o auxílio da ferramenta autotools. Isto gera familiaridade dos usuários com o *AvailableD*, tornando sua instalação e configuração mais simples e fáceis.

2. **Gerenciamento (localidade)**: o *AvailableD* é destinado a ser utilizado em *firewalls*, onde é realizado todo o gerenciamento da solução, de forma centralizada.

3. **Sistemas operacionais nos servidores monitorados**: para realizar o monitoramento, o *AvailableD* utiliza *plugins*, que determinam a situação de disponibilidade dos serviços através de suas interfaces. Dessa maneira, o sistema operacional utilizado nos servidores torna-se transparente e o monitoramento pode ser realizado independentemente do sistema operacional, sendo necessário apenas a pilha de protocolos TCP/IP.

4. **Balanceamento de carga**: o *AvailableD* não possibilita a realização de balanceamento de carga.

5. **Focada em *clusters***: os ambientes alvo do *AvailableD* são redes locais convencionais, sem a existência de *clusters*.

6. **Intrusividade**: desenvolvido para ser utilizado em *firewalls*, o *AvailableD* não faz necessárias alterações na configuração da rede nem instalação de componentes nos servidores monitorados.

Na Tabela 5.1 estão sintetizados os aspectos que foram avaliados nas soluções Linux-HA, LVS, Ultra Monkey e *AvailableD*. Através da tabela, percebe-se que o *AvailableD* possui como pontos positivos a facilidade de instalação e configuração, baixa intrusividade, gerenciamento centralizado e possibilidade de monitorar os serviços em servidores com diversos sistemas operacionais. Como ponto negativo, o *AvailableD* é incapaz de realizar balanceamento de carga. É importante salientar que o ambiente alvo do *AvailableD* é diferente das outras soluções, que foram desenvolvidas para utilização em *clusters* e não em redes locais convencionais.

Tabela 5.1 – Comparação entre o AvailableD, Linux-HA, LVS e Ultra Monkey

	Linux-HA	**LVS**	**Ultra Monkey**	**AvailableD**
Complexidade de instalação	Baixa	Alta	Alta	Baixa
Complexidade de configuração	Alta	Média/Alta	Alta	Baixa
Gerenciamento (localidade)	Distribuído	Centralizado	Distribuído	Centralizado
SOs nos servidores monitorados	GNU/Linux	Qualquer um*	Qualquer um*	Qualquer um*
Balanceamento de carga	Não	Sim	Sim	Não
Focada em *clusters*	Sim	Sim	Sim	Não
Intrusividade	Alta	Média/Alta	Média/Alta	Baixa

Fonte: do autor

* Com suporte à pilha de protocolos TCP/IP. Depende do modo de roteamento utilizado também.

6 Considerações Finais

A alta disponibilidade de serviços e sistemas vem sendo alvo constante da maioria das organizações e instituições. Os clientes estão cada vez mais exigentes e, consequentemente, menos tolerantes a problemas e indisponibilidades. Logo, torna-se crítico trabalhar no sentido de melhorar os índices de disponibilidade dos serviços prestados.

Através da análise de ambientes de redes locais, foi possível identificar técnicas que auxiliam na missão de aumentar níveis de disponibilidade dos serviços. A replicação de serviços é um exemplo de recurso que possibilita garantir a continuidade dos serviços pela utilização de redundâncias. Entretanto, existem problemas que precisam ser tratados, como o fato de muitos clientes serem estaticamente configurados (apontando para uma réplica em específico) e o alto tempo de detecção e utilização das réplicas, como é o caso de muitos clientes DNS.

O objetivo do trabalho foi atacar o problema de uso das réplicas dos serviços pelos clientes através da proposição de uma solução simples, de fácil instalação e com bons índices de usabilidade. A ferramenta projetada tem como alvo redes locais que utilizam *firewalls* GNU/Linux como interfaces de acesso aos serviços da rede. A arquitetura resultante, baseada em utilização de *plugins* e sistemas externos, garantiu um alto nível de extensibilidade, adaptabilidade e tudo de uma maneira simples, utilizando o conceito de acoplamento (*plugin*).

O esforço especial na modelagem da configuração e planejamento da maneira de instalação, assumindo o perfil dos usuários potenciais da aplicação (administradores de redes), resultaram na adoção de convenções que são comuns no nicho de atuação dos usuários. Isso leva a uma imediata familiaridade com ferramenta e maiores níveis de aceitabilidade por parte dos usuários.

A maneira como foi projetada também considerou a intrusividade para implantação, o que influencia diretamente no nível de aceitabilidade. Desenvolvida para ser utilizada em *firewalls*, não há necessidade de alteração na configuração da rede, já que estes possibilitam a execução de re-roteamento de pacotes, uma ação executada a partir do próprio *firewall*, sem necessidade de alterar a estrutura da rede nem de instalar aplicações ou reconfigurar as máquinas monitoradas.

Os testes realizados com o protótipo implementado possibilitaram a validação de seu funcionamento e a mensuração do desempenho da solução. Através de tempos mais curtos entre as verificações de disponibilidade dos serviços, é possível atingir maiores índices de disponibilidade, reduzindo-se a duração dos períodos de indisponibilidade dos serviços percebidos pelos clientes. Em contrapartida, tempos menores entre as verificações, resultam em maior utilização da UCP do *firewall* e aumenta, em pequena escala, o tráfego de rede gerado. A quantidade de

memória alocada, no entanto, aumenta somente de acordo com o número de monitoramentos realizados.

A partir da ferramenta desenvolvida, da análise de suas características, e dos testes realizados, que permitiram verificar e validar o funcionamento e o desempenho da solução, é possível concluir que a solução conseguiu atingir seus objetivos, tanto de cunho funcional como não-funcional.

Como trabalhos futuros, pretende-se resolver o problema de aglomeramento das verificações de disponibilidade, desenvolver ou adaptar *plugins* notificadores, implementar o suporte ao sistema de *log* syslog, implementar a funcionalidade de recarregar a configuração sem precisar parar o *daemon*, criar pacotes de instalação para distribuições GNU/Linux e disponibilizar na internet.

Referências Bibliográficas

BLACK, T. L. Comparação de ferramentas de gerenciamento de redes. *Universidade Federal do Rio Grande do Sul*, 12 2008.

CLUSTERLABS. *Pacemaker*. 2011. Último acesso em maio de 2011. Disponível em: <http:/-/clusterlabs.org/>.

DRDB. *DRDB*. 2011. Último acesso em maio de 2011. Disponível em: <http://www.drbd.org-/>.

HAPROXY. *HAProxy*. 2011. Último acesso em maio de 2011. Disponível em: <http://haproxy-.1wt.eu/>.

HEIN, J. O verdadeiro grande irmão. *Linux Magazine*, jun. 2007.

HORMAN, S. *Ldirectord*. 2011. Último acesso em maio de 2011. Disponível em: <http:/-/horms.net/projects/ldirectord>.

KEEPALIVED. *Keepalived*. 2011. Último acesso em maio de 2011. Disponível em: <http:/-/www.keepalived.org/>.

LINUX-HA. *Linux-HA*. 2011. Último acesso em maio de 2011. Disponível em: <http://www-.linux-ha.org/>.

LVS. *Linux Virtual Server*. 2011. Último acesso em maio de 2011. Disponível em: <http:/-/www.linuxvirtualserver.org/>.

NAGIOS1. *NAGIOS*. 2011. Último acesso em maio de 2011. Disponível em: <http://www-.nagios.org/>.

NAGIOS2. *NAGIOS*. 2011. Último acesso em maio de 2011. Disponível em: <http://nagios-.sourceforge.net/>.

PARZIALE, L. et al. *Achieving High Availability on Linux for System z with Linux-HA Release 2*. [S.l.]: RedBooks, 2009.

SCHILLI, M. Plugando no nagios. *Linux Magazine*, jun. 2007.

ULTRAMONKEY. *Ultra Monkey*. 2011. Último acesso em maio de 2011. Disponível em: <http://www.ultramonkey.org/>.

WIKIPEDIA-CLUSTER. *Computer Cluster*. 2011. Último acesso em dezembro de 2011. Disponível em: <`http://en.wikipedia.org/wiki/Computer_cluster`>.

ZHANG, W. Linux virtual server for scalable network services. *Ottawa Linux Symposium 2000*, 2000.

APÊNDICE A – Arquitetura do Linux-HA

Linux-HA é composto basicamente por 3 componentes:

1. **Heartbeat:** provê a infraestrutura para o cluster. Responsável por garantir a comunicação entre os nós.
2. **Pacemaker:** gerenciador dos recursos do cluster.
3. **Agentes de recursos:** programas escritos para controlar os recursos do cluster. Estes programas contém instruções para iniciar, parar e monitorar recursos.

A arquitetura do Linux-HA é composta de 4 camadas, descritas a seguir:

1. **Camada de infraestrura e mensagens:** é a camada responsável pela troca de mensagens entre os nós do cluster. Composta pelo heartbeat, realiza o controle da situação de cada membro do cluster através de trocas de mensagens. Cada nó envia um sinal ”vital” dizendo ”eu estou vivo” a cada intervalo (configurável) de tempo. Se um nó parar de enviar seu sinal ”vital”, os outros o considerarão ”morto” e um tratamento de failover será iniciado. Torna-se claro que o meio de comunicação entre os nós deve ser altamente confiável, pois falhas neste meio podem acarretar uma má comunicação e, consequentemente, membros que estão operacionais podem ser impedidos de atender requisições. (PARZIALE et al., 2009)
2. **Camada de filiação:** é responsável por calcular a conectividade e realizar a sincronização de informações entre os nós do cluster. Esta camada age em cima das informações recebidas da camada de mensagens e fornece uma visão geral da topologia do cluster para as camadas superiores. Nesta camada temos o Pacemaker. (PARZIALE et al., 2009)
3. **Camada de alocação de recursos:** é onde todas as regras e informações a respeito do cluster são estabelecidas e armazenadas. Esta é a camada administrativa, é o local onde as informações sobre a situação atual do cluster são processadas a fim de gerar ações. Sendo assim, todas as falhas e eventos importantes que ocorrem em quaisquer nós chegarão como informação até esta camada. O processamento desta informação pode definir ações a

serem tomadas para os eventos ocorridos no cluster. Por exemplo, quando for detectado o mau funcionamento de um nó ”x” no cluster, os componentes desta camada irão verificar as regras definidas para este nó e aplicá-las. O Pacemaker é o responsável por esta camada. (PARZIALE et al., 2009)

4. **Camada de recursos:** é a camada que contém os agentes de recursos. É esta a camada responsável por interagir diretamente com os recursos do cluster. Por exemplo, se for identificado que um servidor DNS está desligado, mas há comunicação para este servidor, a camada de alocação de recursos pode definir que este servidor deve ser ligado. Então, esta ação é passada para a camada de recursos que, por sua vez, aciona o agente de recurso específico do DNS e tenta ligá-lo. (PARZIALE et al., 2009)

A figura A.1 ilustra a arquitetura do Linux-HA.

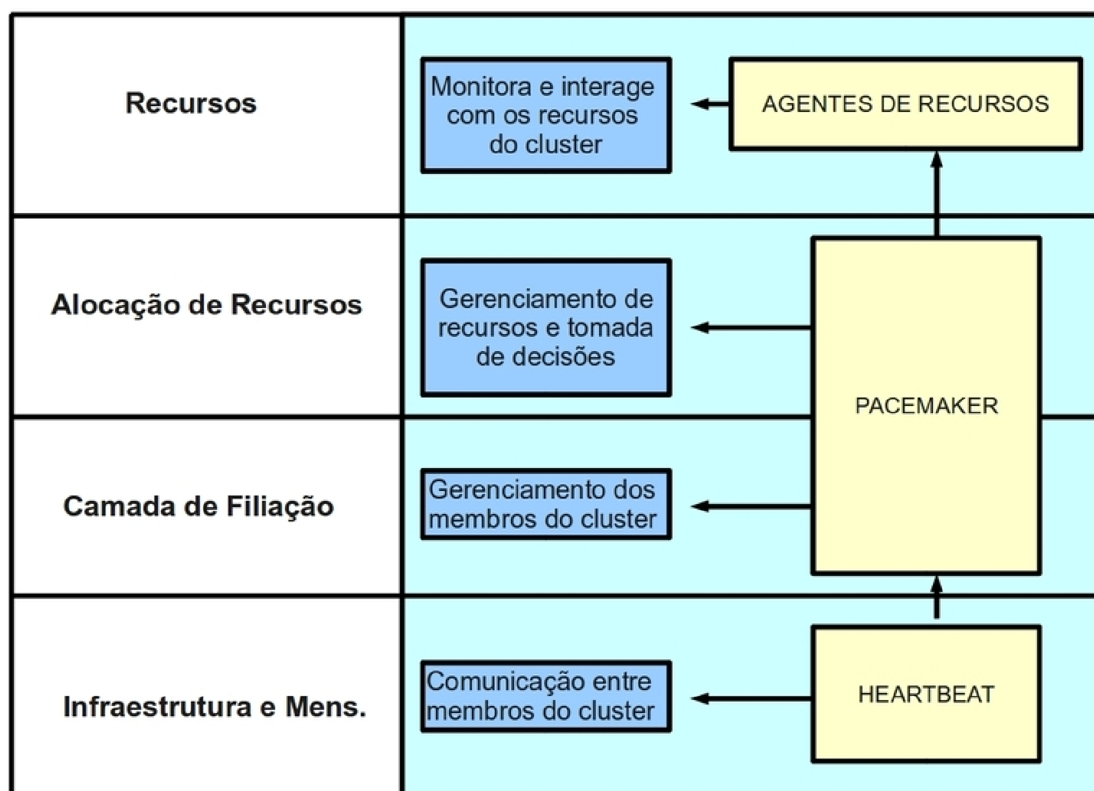


Figura A.1 – Arquitetura do Linux-HA

Em resumo, o Linux-HA provê a comunicação e gerencia os membros do cluster, define ações e controla os recursos.

APÊNDICE B – Instalação, funcionamento e configuração do Linux-HA

Para utilizar o Linux-HA, é necessário que o mesmo seja instalado em todos os nós do cluster. Sendo assim, ele monitora os recursos de cada nó e mantém as informações atualizadas. Sua instalação pode ser realizada de duas maneiras. A mais fácil é utilizar os pacotes pré-construídos disponíveis para a distribuição linux que estiver utilizando (se existirem), mas também é possível fazer o download de seu código fonte e compilá-lo manualmente.

Entre todos os nós do cluster, existe um especial, denominado coordenador (LINUX-HA, 2011). O coordenador é o único que pode definir ações a nível de cluster, ou seja, que afetem a configuração do cluster. Por exemplo, somente o coordenador pode definir a ação de fazer fencing em um nó que apresentou falha em um de seus serviços. Também é de responsabilidade do coordenador iniciar o processo de sincronização de informações entre os membros do cluster. Quando ocorrer algum problema que torne o nó coordenador indisponível, os demais membros iniciam uma eleição, que irá determinar o novo coordenador.

A alta disponibilidade é alcançada pelo Linux-HA através do monitoramento dos recursos do cluster e tratamento de falhas. Um exemplo de tratamento de falha pode ser, por exemplo, iniciar um serviço que era prestado por um nó failover em um outro servidor do cluster. Dessa maneira, o serviço se mantém disponível e o downtime é reduzido.

Uma caraterística importante salientar no Linux-HA é o suporte a quorum, o que ajuda a evitar o cenário de split-brain em caso de falhas nos meios de comunicação.

A configuração do Linux-HA não é centralizada. Cada componente possui sua própria configuração.

Configuração do Heartbeat

O arquivo de configuração do Heartbeat chama-se ha.cf. Neste arquivo são listados os nós do cluster, a topologia de comunicação e quais características da configuração estão liberadas.

(LINUX-HA, 2011)

Sua sintaxe de declarações é bastante simples, podendo cada declaração ser genericamente representada como: parametro_configuracao valor1 [valor2] ... [valorn]

Os comentários consistem em todo o conteúdo da linha contido após um caracter '#'. A lista de opções de configuração disponíveis pode ser encontrada em `http://linux-ha.org/ha.cf`.

Configuração do Pacemaker

O arquivo de configuração do Pacemaker chama-se cib.xml e, como pode-se perceber, utiliza notação xml. Este arquivo divide-se em duas seções principais, sendo elas:

1. **Configuração (configuration):** contém todas as informações a respeito da configuração do cluster, como lista de recursos, locais de alocação de cada recurso, opções de configuração e ações a serem tomadas de acordo com a situação atual do cluster. A seção de configuração divide-se em 4 partes:
   - (a) **Opções de configuração (crm_config):** aqui são definida opções globais para o cluster. Um exemplo seria a opção "symetric-cluster" que determina se os recursos podem ou não ser executados em qualquer nó, o que caracterizaria uma configuração ativo/ativo.
   - (b) **Nós (nodes):** declaração e descrição dos nós do cluster. (CLUSTERLABS, 2011)
   - (c) **Recursos (resources):** declaração dos recursos disponíveis em cada nó do cluster e definição dos agentes de recursos que irão manipulá-los. (CLUSTERLABS, 2011)
   - (d) **Relacionamentos entre recursos e definição de restrições (constraints):** aqui são definidas várias regras necessárias para o equilíbrio e perfeito funcionamento do cluster (CLUSTERLABS, 2011). Esta parte da configuração também é responsável por estabelecer as ações que podem ser tomadas e quando podem ser tomadas. Exemplos de declarações contidas nesta seção: ordem de inicialização dos recursos (necessário para recursos que dependem de outros), nós sobre os quais cada recurso pode ser executado e ações para caso de falhas.
2. **Situação (status):** contém o histórico de cada recurso oferecido por cada nó do cluster (CLUSTERLABS, 2011). Através dos dados contidos nesta seção, o Pacemaker constrói a situação atual do cluster, e assim consegue determinar a execução de ações. O Heartbeat é o responsável por alimentar e manter atualizadas estas informações.

Uma configuração vazia do cib.xml seria assim:

```
<cib generated=true admin_epoch=0 epoch=0 num_updates=0 have-quorum=false>
  <configuration>
    <crm_config/>
    <nodes/>
    <resources/>
    <constraints/>
  </configuration>
  <status/>
</cib>
```

O cib.xml é um arquivo que não deve ser alterado manualmente (CLUSTERLABS, 2011). Para qualquer alteração, deve ser utilizado o cibadmin, uma ferramenta comand line que realiza as alterações e as faz surtir efeito imediatamente.

A configuração do Pacemaker deve estar consoante com a do Heartbeat, caso contrário, podem possuir visões diferentes da estrutura e configuração do cluster, o que acarretaria problemas de comunicação entre eles. Isto torna a implantação da solução Linux-HA bastante complexa e a gerência pouco eficiente, já que algumas atualizações deverão ser executadas nos dois programas.

APÊNDICE C – Algoritmos de escalonamento do LVS

O balanceador de carga ao receber uma solicitação de um cliente, deve escolher um servidor para atender a esta solicitação. Para determinar qual será este servidor, existem alguns algoritmos suportados pelo IPVS que procuram distribuir da melhor forma as requisições recebidas.

- **Round-robin:** este é um algoritmo de rodízio onde cada solicitação será enviada para um servidor diferente. Um mesmo cliente pode ser direcionado a servidores diferentes a cada solicitação. (ZHANG, 2000)
- **Weighted round-robin:** é o mesmo round robin, porém com pesos determinados para cada servidor (ZHANG, 2000). O peso significa o poder de processamento do servidor. Quanto maior o peso atribuído, maior será o número de requisições a serem atendidas pelo servidor. Pesos maiores também irão se traduzir em prioridade de utilização, ou seja, servidores com pesos altos serão utilizados primeiro.
- **Least-connection:** direciona as requisições para os servidores que possuírem o menor número de conexões ativas. Quando os servidores possuem capacidades diferentes de processamento este algoritmo não funciona muito bem, pois ele divide igualmente as requisições, sem considerar outros aspectos (ZHANG, 2000).
- **Weighted least-connection:** funcionamento semelhante ao least-connection, porém, há a possibilidade de atribuir pesos para cada servidor (ZHANG, 2000). Quanto maior o peso, maior será a porcentagem de requisições a serem atendidas pelo servidor.
- **Locality-based least-connection:** semelhante ao weighted least-connection, porém, é verificado se o servidor está sobrecarregado (número de conexões maior que seu peso). Se ele estiver sobrecarregado, a requisição será enviada para um outro servidor que não esteja. Se todos os servidores estiverem sobrecarregados, então, o que foi inicialmente selecionado para atender a solicitação é quem de fato o fará.
- **Locality-Based Least-Connection com replicação:** difere-se do Locality-based least-connection no fato de que quando todos os servidores estiverem sobrecarregados, o mais

sobrecarregado de todos será removido da lista de servidores a serem selecionados.

- **Destination Hashing:** atribui conexões para os servidores utilizando uma tabela hash estática para os endereços de destino.
- **Source Hashing:** atribui conexões para os servidores utilizando uma tabela hash estática para os endereços de origem.
- **Menor Delay esperado:** encaminha a conexão para o servidor que possuir menor delay esperado para o serviço. O delay esperado é calculado através da fórmula: (C + 1) / U, onde C é o número de conexões atualmente ativas para o servidor e U é sua capacidade de processamento corrigida para o serviço.

Um problema que pode ocorrer com o balanceamento de carga é quando existe a necessidade de conexão persistente. Exemplo de aplicações que necessitam de persistência são aquelas que criam uma sessão após a autenticação, o que é muito comum em serviços web. Quando a sessão é iniciada em um servidor, ela somente existe nele. Portanto, se o cliente for redirecionado para outro servidor, outra sessão deverá ser iniciada.

Para resolver este problema, o LVS permite configurar os serviços com conexões persistentes. Quando um cliente acessa um serviço persistente, um registro com a informações do cliente e servidor é inserido em uma tabela. Então, a cada nova conexão do cliente, o mesmo servidor será responsável por atendê-lo, independendo do algoritmo de escalonamento utilizado. Para impedir problemas com o balanceamento de carga, existe um timeout para os registros desta tabela. Este timeout é configurável pelo usuário e após ser atingido, o registro é automaticamente excluído.

APÊNDICE D – Métodos de roteamento suportados pelo LVS

O LVS possui 3 métodos de roteamentos, que podem ser utilizados simultaneamente, embora não seja usual tal configuração.

NAT

Este é o método mais simples e mais fácil de configurar. Para utilizar NAT com o LVS, é necessário apenas que o diretor possua um IP virtual (LVS, 2011), o qual será utilizado para realizar a comucação entre os servidores do cluster e os clientes.

A cada pacote recebido, o LVS verifica se já existe alguma conexão estabelecida para ele em sua tabela hash de conexões (LVS, 2011). Se ainda não existe, um servidor é eleito para atender a requisição e um novo registro é salvo na tabela, pois é através dela que o diretor mantém os pacotes de uma conexão já estabelecida sendo enviados sempre para um mesmo servidor. Após, faz a tradução NAT, alterando endereço e porta de destino contidos no pacote IP para os do servidor que irá atender a solicitação. Os servidores possuem como gateway padrão o próprio diretor, portanto, as respostas retornarão a ele após terem sido processadas.

Quando o diretor receber a resposta do servidor, outra tradução é realizada, sendo agora colocado o IP virtual como endereço de origem. Sempre que uma conexão terminar ou atingir seu timeout, seu registro é eliminado da tabela hash.

A figura D.1 detalha o funcioamento do LVS com NAT.

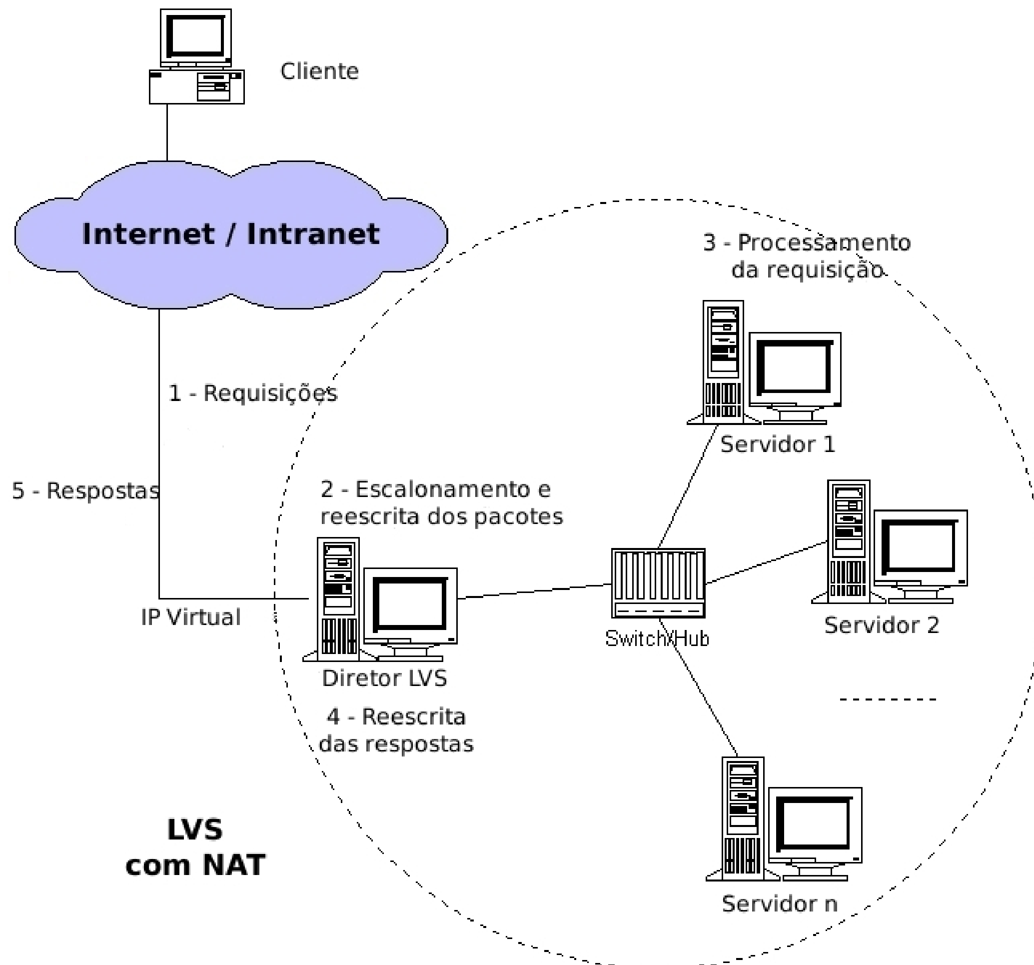


Figura D.1 – LVS com NAT. Figura adaptada de (LVS, 2011)

Utilizando esta opção de roteamento, é possível que qualquer máquina que possua a pilha de protocolos TCP/IP seja um servidor LVS.

Roteamento direto

No roteamento direto, as requisições passam pelo diretor, que escolhe um servidor para realizar o atendimento, grava a conexão na tabela hash e redireciona os pacotes para este servidor (LVS, 2011). O servidor por sua vez, conecta-se diretamente ao cliente, utilizando sua própria tabela de roteamento. Para fazer o roteamento direto é preciso configurar em todos os servidores o mesmo IP virtual configurado no diretor, pois o procedimento realizado quando do recebimento de pacotes pelo diretor é somente alterar o endereço físico de destino (MAC) para o do servidor escolhido. Isto gera a necessidade de ambos estarem no mesmo meio físico, pois somente desta maneira é possível recuperar o MAC do servidor.

Utilizando esta configuração, pode ocorrer um problema relacionado com as requisições

ARP. O problema se dá justamente pelo fato de todas as máquinas possuírem o mesmo IP virtual, o que pode causar das requisições irem diretamente para os servidores, sem intervenção do diretor, ou seja, sem balanceamento de carga. Por este motivo, o IP virtual deve preferencialmente ser configurado em uma interface de loopback, que na maioria dos sistemas operacionais não responde a requisições ARP. Infelizmente, o único sistema operacional que apresentou este problema foi justamente o linux. Porém, já existe um patch para ser aplicado no kernel que resolve esta situação, só que isto exige a recompilação do mesmo.

A figura D.2 detalha o funcionamento do LVS com roteamento direto.

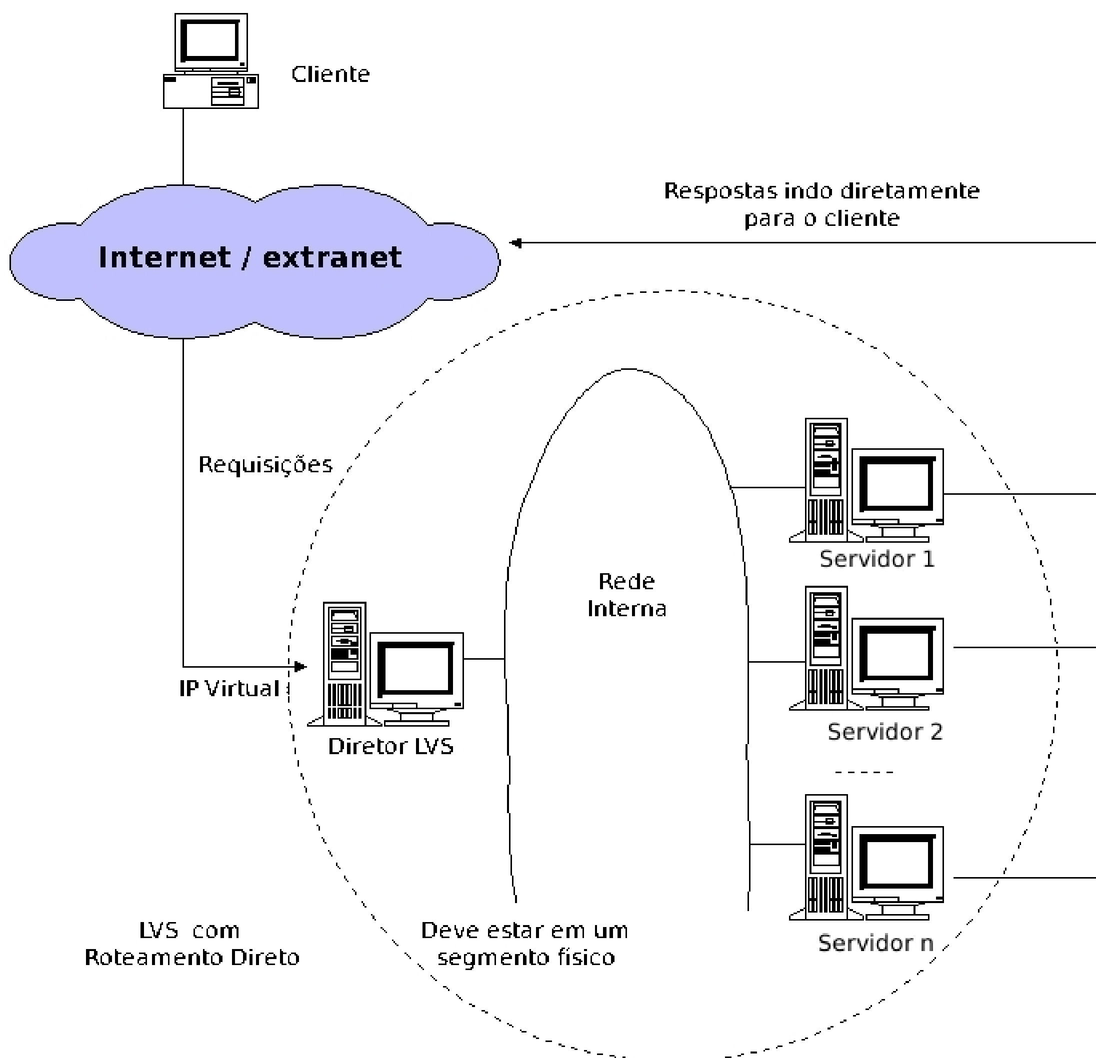


Figura D.2 – LVS com Roteamento Direto. Figura adaptada de (LVS, 2011)

O LVS com roteamento direto permite praticamente qualquer sistema operacional rodando nos servidores, desde que seja possível resolver o problema com o ARP.

Tunelamento IP

Tunelamento IP é uma técnica que nem todos sistemas operacionais suportam atualmente. Consiste em encapsular um pacote IP dentro de outro, gerando um pacote denominado IPIP (LVS, 2011). Neste âmbito, a diferença do tunelamento IP para o NAT é que a requisição original (cliente/diretor) não é alterada, somente encapsulada dentro de uma nova requisião no sentido diretor / servidor.

Cada requisição que chega no diretor é verificada, a fim de determinar se a conexão já foi ou não estabelecida para um servidor real. Caso ainda não tenha sido, escolhe um servidor para atendê-la, insere o registro na tabela hash, encapsula a requisição em um pacote IP e envia para o servidor. No outro lado, o servidor recebe o pacote em uma interface que seja capaz de manipular o tunelamento, desencapsula e conecta-se diretamente ao cliente, utilizando sua própria tabela de rotas. A configuração do IP virtual no diretor e nos servidores é idêntica a configuração com roteamento direto, e existe o mesmo problema com as requisições ARP para ser tratado. A única diferença, é que o IP virtual deve ser configurado em uma interface que manipule pacotes encapsulados.

A figura D.3 demostra o funcionamento do LVS com tunelamento IP.

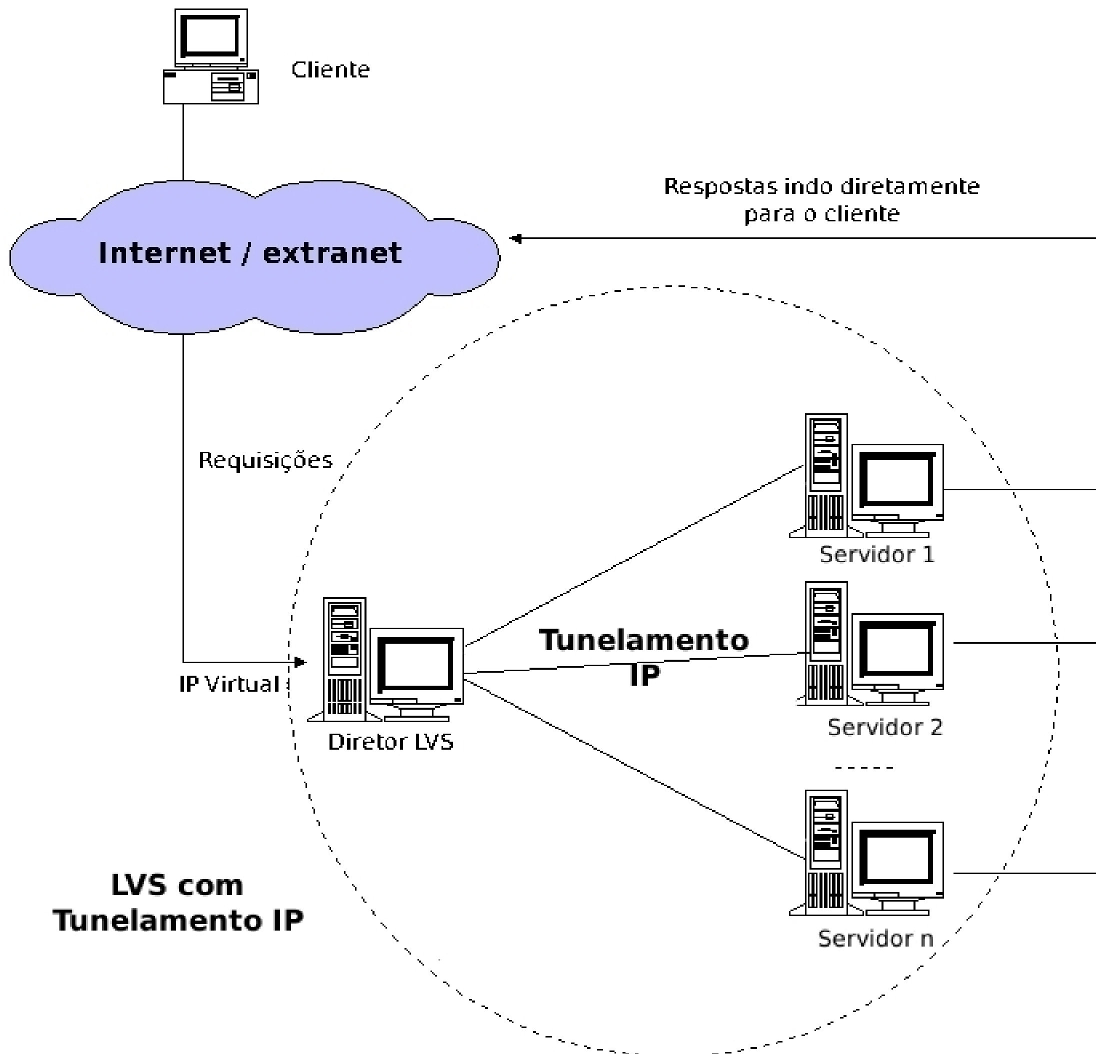


Figura D.3 – LVS com Tunelamento IP. Figura adaptada de (LVS, 2011)

O tunelamento IP traz a grande vantagem de não necessitar que os diretores e servidores estejam em uma mesma rede local, uma vez que uma nova requisição é gerada em vez de somente modificar as recebidas. Os sitemas linux suportam tunelamento IP.

APÊNDICE E – Arquitetura do Ultramonkey

Sua arquitetura é muito semelhante a do LVS, com o único diferencial de existir um diretor de backup, que é monitorado pelo Linux-HA.

A figura demonstra a arquitetura do Ultra Monkey.

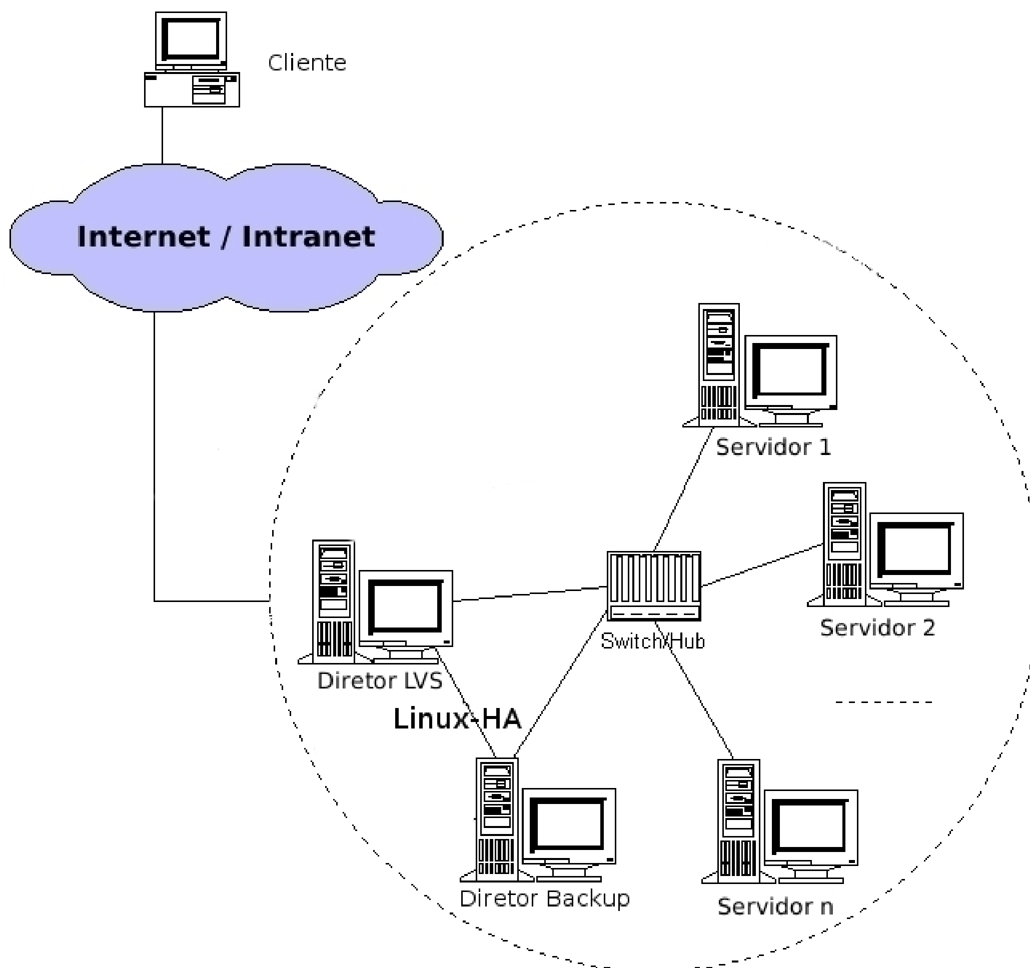


Figura E.1 – Arquitetura do Ultra Monkey

APÊNDICE F – Documento de Requisitos

Documento de requisitos do sistema de re-roteamento automático e transparente de pacotes em firewalls para aumentar a disponibilidade de serviços de rede com redundância implementada AvailableD

Versão 0.4

Sobre este Documento

Este documento consiste de uma adaptação do documento de requisitos do sistema Methodology Explorer, que foi utilizado como modelo e pode ser encontrado no endereço:
http://www.cin.ufpe.br/~mexplorer/metodologia/requisitos/documentoRequisitos.doc.

Histórico de Alterações

Data	Versão	Descrição	Autor
16/06/11	0.1	Primeira versão do documento.	Tony F. B. M. Ribeiro
04/07/11	0.2	Revisado por Diego L. Kreutz.	Tony F. B. M. Ribeiro
05/07/11	0.3	Revisado por Tony F. B. M. Ribeiro	Tony F. B. M. Ribeiro
02/12/11	0.4	Ajustes na forma de escrita do documento	Tony F. B. M. Ribeiro

Conteúdo

1. INTRODUÇÃO 6

1.1 Visão geral do documento....6
1.2 Convenções, termos e abreviações....6
1.2.1 Identificação dos requisitos....6
1.2.2 Prioridades dos requisitos....6

2. DESCRIÇÃO GERAL DO SISTEMA 7

2.1 Abrangência....7

3. REQUISITOS FUNCIONAIS (CASOS DE USO) 7

[RF001] Verificação dos servidores e serviços....7
[RF002] Adicionar / remover regras de roteamento de pacotes....8
[RF003] Armazenamento de informações sobre as falhas ocorridas....8
[RF004] Informar problemas aos administradores da rede....8
[RF005] Configuração do sistema....9

4. REQUISITOS NÃO-FUNCIONAIS 9

[NF001] Usabilidade....9
[NF002] Desempenho....9
[NF003] Portabilidade....10
[NF004] Confiabilidade....10
[NF005] Interoperabilidade....10
[NF006] Simplicidade....10

5. REFERÊNCIAS 12

1. Introdução

Este documento especifica os requisitos do sistema de re-roteamento automático e transparente de pacotes em firewalls para aumentar a disponibilidade de serviços de rede com redundância implementada *AvailableD*, fornecendo aos desenvolvedores as informações necessárias para o projeto e implementação, assim como para a realização dos testes e validação do sistema.

1.1 Visão geral do documento

Além desta seção introdutória, as seções seguintes estão organizadas como descrito abaixo.

1. **Seção 2 – Descrição geral do sistema**: apresenta uma visão geral do sistema, caracterizando qual é o seu escopo e descrevendo seus usuários.
2. **Seção 3 – Requisitos funcionais (casos de uso)**: especifica todos os casos de uso do sistema, descrevendo seus fluxos de eventos
3. **Seção 4 – Requisitos não funcionais**: especifica todos os requisitos não funcionais do sistema.
4. **Seção 5 – Referências:** apresenta referências para outros documentos utilizados para a confecção deste documento.

1.2 Convenções, termos e abreviações

A correta interpretação deste documento exige o conhecimento de algumas convenções e termos específicos, que são descritos a seguir.

1.2.1 Identificação dos requisitos

Por convenção, a referência a requisitos é feita através do nome da subseção onde eles estão descritos, de acordo com a especificação a seguir:
[*nome da subseção*]
Os identificadores de requisitos devem ser únicos. A numeração inicia com o identificador [RF001] ou [NF001] e prossegue sendo incrementada à medida que forem surgindo novos requisitos.

1.2.2 Prioridades dos requisitos

Para estabelecer a prioridade dos requisitos, nas seções 3 e 4, foram adotadas as denominações “essencial”, “importante” e “desejável”

5. **Essencial** é o requisito sem o qual o sistema não entra em funcionamento. Requisitos essenciais são requisitos imprescindíveis, que têm que ser implementados impreterivelmente.
6. **Importante** é o requisito sem o qual o sistema entra em funcionamento, mas de forma não satisfatória. Requisitos importantes devem ser implementados, mas, se não forem, o sistema poderá ser implantado e usado mesmo assim.
7. **Desejável** é o requisito que não compromete as funcionalidades básicas do sistema, isto é, o sistema pode funcionar de forma satisfatória sem ele. Requisitos desejáveis podem ser deixados para versões posteriores do sistema, caso não haja tempo hábil para implementá-los na versão que está sendo especificada.

2. Descrição geral do sistema

2.1 Abrangência

O sistema AvailableD tem por objetivo auxiliar na tarefa de melhorar a disponibilidade de serviços de redes locais que possuem redundância implementada, de forma automática e transparente, utilizando como artifício para tal a ativação e remoção de regras de roteamento de pacotes em firewalls que fazem a ponte entre clientes e serviços. O sistema tem como alvo ambientes GNU/Linux, devido a grande relevância destes ambientes na utilização em servidores.

O princípio básico do sistema é a simplicidade de instalação, configuração e uso, a fim de permitir que gerentes de redes menos experientes consigam utilizá-lo sem dificuldades. Seu nicho de aplicação restringe-se a redes locais que possuem uma arquitetura baseada em firewall(s) roteadores como pontos únicos de acesso aos serviços, o que é uma configuração comum em redes locais de pequeno e médio porte.

Os recursos disponibilizados pelo sistema são:

- Verificação da disponibilidade e funcionamento dos servidores e serviços prestados;
- Re-roteamento automático e transparente de pacotes entre servidores que prestem o mesmo serviço quando da falha de alguma das réplicas, bem como a remoção do re-roteamento quando do retorno a normalidade da mesma.
- Armazenamento das estatísticas de re-roteamentos realizados;
- Informar aos administradores da rede sobre falhas ocorridas nos servidores e serviços;
- Configuração do sistema para atender as necessidades específicas de cada usuário.

3. Requisitos funcionais (casos de uso)

[RF001] Verificação dos servidores e serviços

Descrição: Deve ser possível verificar a situação dos servidores e serviços de rede. Após a verificação de cada servidor/serviço, deve ser possível determinar se o mesmo está funcionando corretamente.

Fluxo de execução:

a) O sistema está rodando e possui configuração para verificar algum servidor/serviço.

b) O sistema envia uma requisição para o servidor.

c) O sistema espera a resposta, repeitando um tempo limite de espera.

d) Se a resposta foi recebida, e está de acordo com o esperado, o sistema determina que o servidor/serviço está disponível. Caso o tempo limite de espera seja atingido sem obter resposta, ou a resposta recebida não esteja em conformidade com o esperado, determina que o servidor/serviço está indisponível.

Prioridade: ν Essencial Importante Desejável

[RF002] Adicionar / remover regras de roteamento de pacotes

Descrição: Sempre que identificado que um servidor/serviço torna-se indisponível, deve ser adicionada uma regra de roteamento que redirecione os pacotes destinados a este servidor para um outro que preste o mesmo serviço e que esteja funcional.
Fluxo de execução:

1. *Adição de regras em caso de falha de um serviço/servidor:*

a) O sistema identifica que um serviço está indisponível em um servidor;
b) Se existe um outro servidor no qual o serviço está disponível, adiciona uma regra de roteamento que redireciona os pacotes destinados ao servidor problemático para um servidor disponível.

2. *Remoção de regras em caso de retorno de um serviço/servidor:*

a) O sistema identifica que um serviço voltou a estar disponível em um servidor;
b) Remove a regra de re-roteamento dos pacotes destinados a este servidor.

Prioridade: ν Essencial Importante Desejável

[RF003] Armazenamento de informações sobre as falhas ocorridas

Descrição: O sistema deve gravar informações sobre a ocorrência de falhas nos servidores e serviços, a fim de possibilitar futura análise destas informações pelos administradores da rede. A análise estatística ajuda a descobrir problemas potenciais na rede, como servidores que apresentam altos índices de indisponibilidade.
Fluxo de execução:
a) O sistema identifica algum problema ao verificar um servidor ou serviço.
b) O sistema armazena as informações do servidor/serviço que apresentou problemas durante a verificação.

Prioridade: Essencial Importante ν Desejável

[RF004] Informar problemas aos administradores da rede

Descrição: O sistema deve ser capaz de enviar notificações para os administradores da rede, comunicando problemas encontrados durante a verificação dos servidores e serviços. Isto permite que os administradores tomem providências mais rapidamente para corrigir os problemas encontrados.
Fluxo de execução:
a) O sistema identifica algum problema ao verificar um servidor/serviço.
b) O sistema envia uma notificação aos administradores da rede contendo informações do problema encontrado.

Prioridade: Essencial ν Importante Desejável

[RF005] Configuração do sistema

Descrição: O sistema deve ser configurável, a fim de permitir sua adaptação às diferentes configurações de rede nas quais for utilizado.
Fluxo de execução:
a) O sistema lê sua configuração.
b) O sistema passa a comportar-se de acordo com a configuração lida.

Prioridade: ν Essencial Importante Desejável

4. Requisitos não-funcionais

[NF001] Usabilidade

Sua instalação, configuração e uso devem ser fáceis, com uma alta curva de aprendizado.

Prioridade: ν Essencial Importante Desejável

[NF002] Desempenho

O sistema deve apresentar bom desempenho, principalmente por rodar em firewalls.

Prioridade: ν Essencial Importante Desejável

[NF003] Portabilidade

O sistema não tem foco em portabilidade, sendo destinado a ambientes GNU/Linux. Porém, seria desejável ser portável para outras arquiteturas.

Prioridade: Essencial Importante ν Desejável

[NF004] Confiabilidade

O sistema deve prover meios de impedir a ocorrência de inconsistência de operação, principalmente devido a configurações inadequadas. O sistema jamais deve prejudicar o funcionamento da rede e muito menos prejudicar a máquina que o executa. O sistema deve ter um bom nível de confiabilidade.

Prioridade: ν Essencial Importante Desejável

[NF005] Interoperabilidade

O sistema deve ser capaz de verificar qualquer servidor/serviço, independente do sistema operacional que o mesmo utiliza, pois utilizará as interfaces dos serviços.
O ambiente de execução dos módulos de verificação dos serviços será restrito a ambientes GNU/Linux.

Prioridade: ν Essencial Importante Desejável

[NF006] Simplicidade

O sistema deve ser simplista, principalmente quanto a seu uso, configuração e gerenciamento. Simplista refere-se a restringir-se ao extremamente essencial, o que reduz a carga de conhecimento necessário e, consequentemente, a complexidade.

Prioridade: ν Essencial Importante Desejável

5. Referências

1. Documento de requisitos do Methodology Explorer, disponível em: http://www.cin.ufpe.br/~mexplorer/metodologia/requisitos/documentoRequisitos.doc, acessado em 10/05/2011.
2. PRESSMAN, Roger S. *Engenharia de Software.* McGraw-Hill, 2006.
3. SOMMERVILLE, Ian. *Software Engineering*. 8ª ed. Addison Wesley, 2006.
4. WIKIPEDIA. Engenharia de Requisitos. [Online]. Disponível em: http://pt.wikipedia.org/wiki/Engenharia_de_requisitos, acessado em 15/06/2011

APÊNDICE G – Documento de Projeto

Documento de projeto do sistema de re-roteamento automático e transparente de pacotes em firewalls para aumentar a disponibilidade de serviços de rede com redundância implementada AvailableD

Versão 0.4

Sobre este Documento

Este documento consiste de uma adaptação do documento de requisitos do sistema *Methodology Explorer.* O documento original pode ser encontrado no endereço: http://www.cin.ufpe.br/~mexplorer/metodologia/requisitos/documentoRequisitos.doc.

Histórico de Alterações

Data	Versão	Descrição	Autor
30/06/11	0.1	Primeira versão do documento.	Tony F. B. M. Ribeiro
04/11/11	0.2	Alterações após revisão por Diego L. Kreutz	Tony F. B. M. Ribeiro
03/12/11	0.3	Atualização do documento após implementação do sistema.	Tony F. B. M. Ribeiro
15/12/11	0.4	Alterações após revisão completa do documento.	Tony F. B. M. Ribeiro

Conteúdo

INTRODUÇÃO
6

Visão geral do documento......6

1. VISÃO GERAL DO SISTEMA
7

1.1. Problematização e Contextualização......7
1.2. Arquitetura Geral......8
1.4. Cenários de Aplicação Prática......9
1.3. Funcionamento do Sistema......10
1.5. Tecnologias e Requisitos de Sistema......13
1.6. Documentação e Tratamento de Ajuda ao Usuário......13

2. PROJETO LÓGICO E ARQUITETURAL
15

2.1. Especificação Funcional......15
2.2. Arquitetura......17

3. PROJETO DE COMPONENTES, DADOS E INTERFACES
20

3.1. Configuração......20
3.2. Log......32
3.3. Notificação......34
3.5. Monitoramento......39
3.6. Executor de Ações......42
3.7. Integração de Componentes......44

4. PADRÕES DE CODIFICAÇÃO
45

4.1. Nomes e Declaração de Variáveis......45
4.2. Nomes e Declaração de Funções......45
4.3. Comando de controle de fluxo......46
4.4. Chamadas de funções......46
4.5. Operadores......47
4.6. Identação......47
4.7. Comentários e Documentação do Código......47
4.8. Demais padrões......47

REFERÊNCIAS
49

Introdução

Este documento especifica a arquitetura, funcionalidades e os detalhes para implementação do sistema *AvailableD*.

Visão geral do documento

Além desta seção introdutória, as seções seguintes estão organizadas como descrito abaixo.

Visão Geral do Sistema – define aspectos gerais do sistema e o seu funcionamento de forma resumida.

Projeto Lógico e Arquitetural – definição do funcionamento e da arquitetura do *AvailableD*.

Projeto de Componentes, Dados e Interfaces – especificação detalhada dos componentes, tecnologias utilizadas, estruturas de dados, definição das interfaces de comunicação e integração dos componentes.

Padrões de Codificação – especifica os padrões utilizados na codificação do software.

1. Visão Geral do Sistema

O sistema *AvailableD* é um *daemon* com a finalidade de aumentar a disponibilidade de serviços com redundância implementada, em redes que possuem *firewalls* roteadores como porta de acesso a tais serviços. Para isso, realiza o monitoramento e, ao identificar a indisponibilidade de um serviço, altera as regras de roteamento do *firewall* de forma que os pacotes destinados ao servidor problemático do serviço, sejam redirecionados para um outro servidor do mesmo serviço que esteja disponível. Quando o servidor problemático voltar a funcionar corretamente, as regras do *firewall* são novamente alteradas, removendo-se o redirecionamento realizado.

1.1. Problematização e Contextualização

Em redes locais de organizações concentram-se diversos tipos de sistemas e serviços que são de fundamental importância, tanto para funcionamento da infraestrutura básica (DNS, DHCP, etc), como para para o próprio negócio (HTTP, banco de dados, etc). A indisponibilidade destes serviços consiste em um grande problema, pois, além de gerar insatisfação em seus usuários, pode gerar graves prejuízos para a organização. Há, então, a necessidade de manter níveis mais elevados de disponibilidade de tais serviços.

Grande parte das organizações possuem réplicas de seus serviços, geralmente com a finalidade de *backup*, e como um meio de colocar serviços de volta a ativa de forma mais rápida, em ocasiões de mau funcionamento da réplica principal. O problema é que, mesmo no melhor caso, se não houver o auxílio de uma ferramenta, há um considerável intervalo de tempo entre detecção de um problema e efetivação de uma ação paliativa.

Com vistas neste problema, já foram desenvolvidas várias soluções, tendo entre algumas que se destacam o Linux-HA (http://www.linux-ha.org), o Nagios (http://www.nagios.org/) e o Linux Virtural Server (http://www.linuxvirtualserver.org/), ambas soluções livres. Há varias outra soluções, muitas das quais são pagas.

Mesmo havendo uma enorme gama de sistemas já implementados, não foi encontrado nenhum que combinasse as seguintes características: facilidade de instalação e configuração, simplicidade, gratuito, centralização da configuração e do gerenciamento, baixa intrusividade, dedicado a redes de menor porte e dedicado a utilização em *firewalls* roteadores que fazem o intermédio entre clientes e serviços.

O sistema *AvailableD* tem, portanto, o objetivo de ser a solução que congrega tais características, visando aumentar a disponibilidade de serviços de uma forma gratuita, simples e fácil em ambientes de rede baseados em *firewalls* roteadores, onde há a existência de réplicas dos serviços.

Para atingir tal objetivo, foi constatada a necessidade dos seguintes requisitos funcionais:

1. Verificação da disponibilidade e funcionamento dos servidores e serviços prestados;
2. Re-roteamento automático e transparente de pacotes entre servidores que prestem o mesmo serviço, como forma de contornar falhas nas réplicas. Dessa forma, os serviços mantém-se disponíveis enquanto houver uma réplica funcional.
3. Armazenamento das estatísticas de re-roteamentos realizados, permitindo análise pelos administradores da rede;

4. Informar aos administradores da rede sobre os detalhes de falhas ocorridas nos servidores e serviços, possibilitando a redução do tempo de resolução dos problemas;

5. Capacidade de serem informados os serviços a serem monitorados, suas informações, detalhes para a realização das verificações de disponibilidade, comandos do *firewall* para realizar o re-roteamento e configuração de notificação.

Também foram identificados os seguintes requisitos não funcionais:

1. Usabilidade;
2. Desempenho;
3. Portabilidade (desejável, apenas);
4. Confiabilidade;
5. Interoperabilidade;
6. Simplicidade.

Os requisitos supracitados estão explicados em detalhes no documento de requisitos.

1.2. Arquitetura Geral

A fim de prover as funcionalidades necessárias e contemplar os requisitos não funcionais, a arquitetura geral do sistema foi dividida em módulos:

1. **Configuração:** responsável por ler, gerenciar e armazenar a configuração de todo o sistema. É o responsável por obter as informações de configuração utilizadas pelos demais módulos;
2. **Monitoramento:** realiza as verificações de disponibilidade de serviços através da utilização de *plugins* verificadores. É responsável por determinar a situação dos servidores.
3. **Executor de ações:** executa ações de acordo com os resultados do monitoramento. As ações consistem dos comandos (*plugins*) do *firewall* para ativar ou remover os redirecionamentos de pacotes.
4. **Log:** envia mensagens recebidas para gerenciadores de *log*, responsáveis por realizar o armazenamento destas mensagens.
5. **Notificação:** envia notificações de acordo com a configuração estabelecida. Para emitir as notificações, são utilizados *plugins*.
6. **Integração:** realiza a integração entre os demais módulos, possibilitando a comunicação entre eles. Responsável pela lógica de operação do *AvailableD*.

A figura 1 contém a representação da arquitetura geral do *AvailableD*.

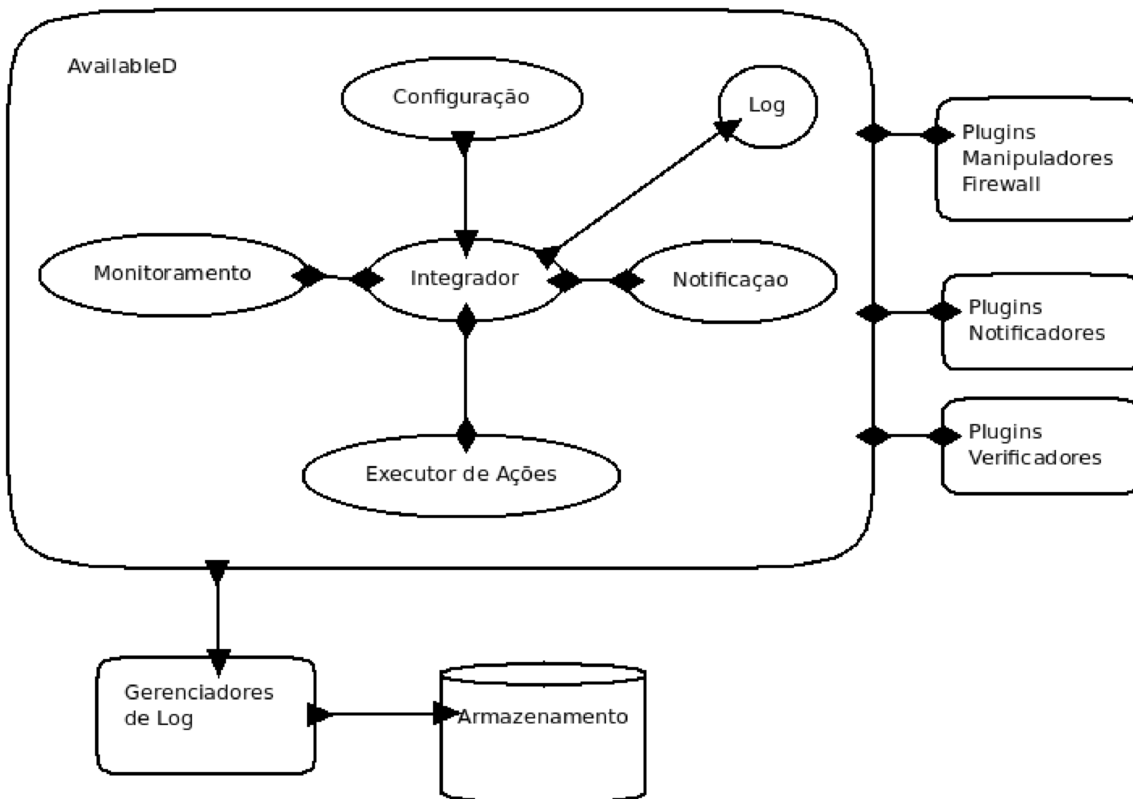


Figura 1: Representação da arquitetura geral do *AvailableD*.

1.4. Cenários de Aplicação Prática

O *AvailableD* tem como alvo redes locais de pequeno e médio porte onde há a presença de *firewalls* roteadores como pontos de acesso aos serviços, o que é uma configuração comum em redes destes portes.

Para melhor entendimento da utilização da solução, vamos tomar como exemplo um cenário típico. Considere uma pequena organização que possui um sítio WEB hospedado em sua rede local. Esta rede é composta por 2 máquinas, sendo que uma delas possui o servidor HTTP principal, e a outra é uma réplica, também capaz de atender as requisições, pois possui uma cópia idêntica das páginas e acessa os mesmos dados. Há também um *firewall*, que é a máquina que possui acesso direto à internet e é a interface para acesso aos serviços da rede local.

Na figura 2 está representada a rede descrita.

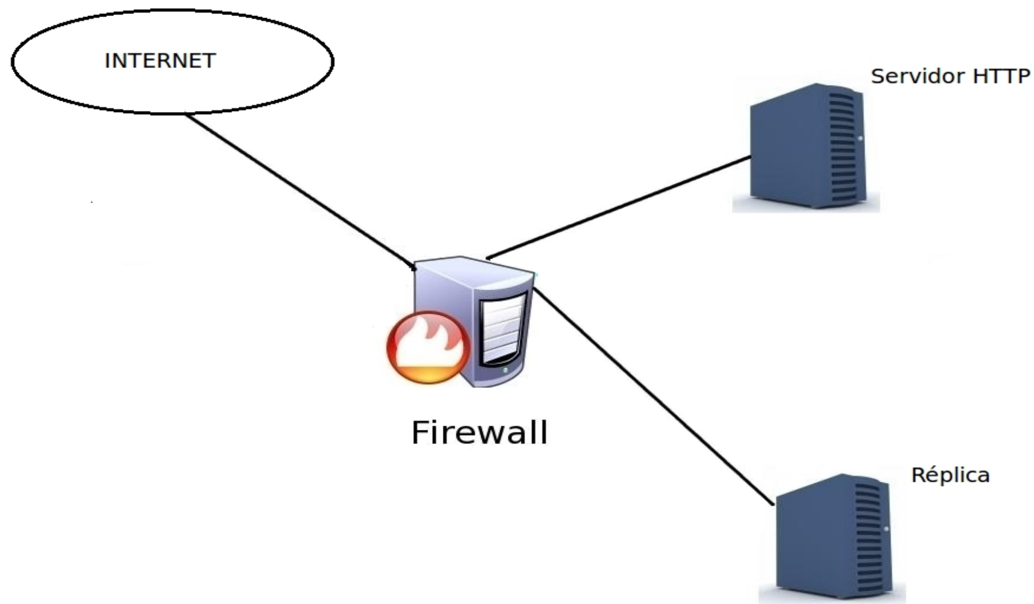


Figura 2: Exemplo de cenário prático de aplicação.

O cenário desta desta organização é propício para a aplicação da solução, pois satisfaz as duas condições principais:

1. O serviço possui redundância implementada;
2. Possui um *firewall* como ponto único de acesso aos serviços da rede.

Isso permite que o sistema seja instalado no *firewall*, monitore os servidores e ative regras de redirecionamento para a réplica que estiver disponível sempre que uma delas apresentar problemas. Dessa maneira, os clientes continuarão acessando o mesmo serviço em outra réplica, de forma totalmente transparente.

1.3. Funcionamento do Sistema

O sistema possui um comportamento bastante simples, podendo ser sintetizado nos seguintes passos:

1. Ao iniciar, carrega sua configuração, na qual estão as informações dos serviços que serão monitorados, as ações a serem executadas em casos de indisponibilidade e de retorno dos servidores (comandos de *firewall*) e informações a respeito de log e notificações;
2. O sistema inicia o monitoramento dos serviços;
3. Sempre que for detectada falha em um servidor, e quando o mesmo voltar a estar funcional, executa as ações configuradas (adição ou remoção de redirecionamento de pacotes), realiza log das informações e emite notificação (se configurado para tal).

A figura 3 apresenta o fluxo de execução simplificado do *AvailableD*.

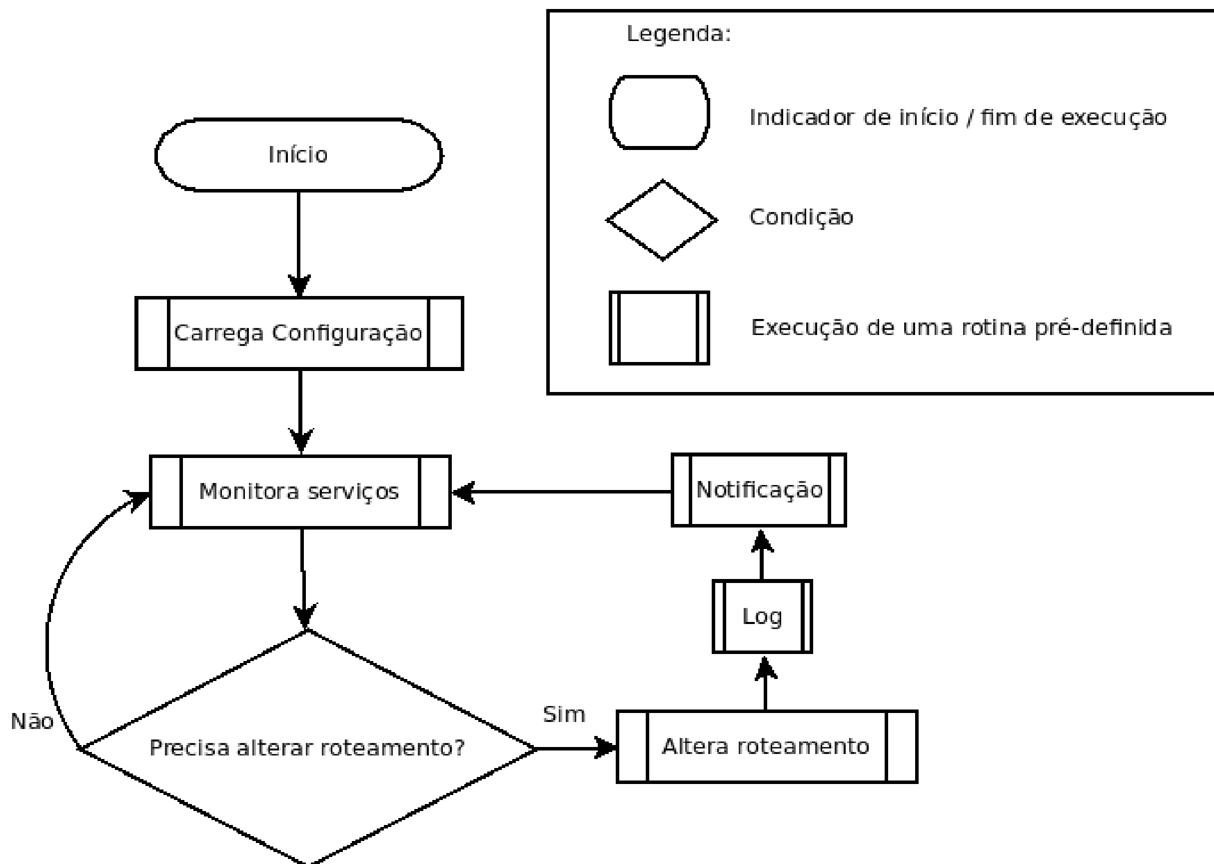


Figura 3: Fluxo de execução simplificado do *AvailableD*.

Perceba que segundo o fluxo apresentado, o programa nunca irá encerrar sua execução. Por tratar-se de um *daemon,* este comportamento é normal. Para parar o sistema, é necessário a execução de um comando de parada do *daemon*.

As figuras 4 e 5 ilustram em um cenário prático o que acontece com as requisições quando um dos servidores falha. Percebe-se que desta maneira, a disponibilidade do serviço é assegurada.

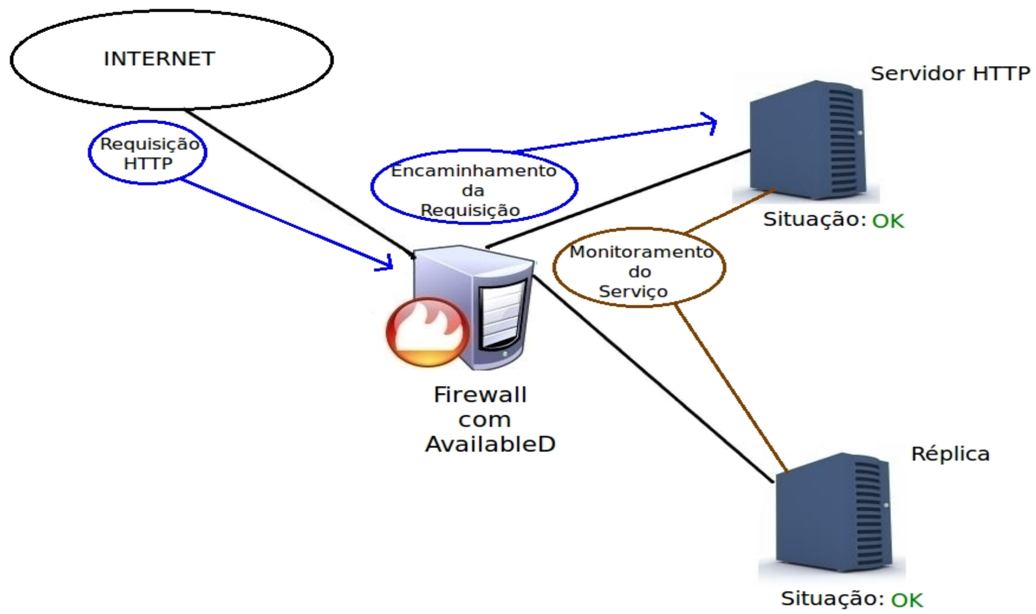


Figura 4: Funcionamento em ocasião de normalidade dos servidores.

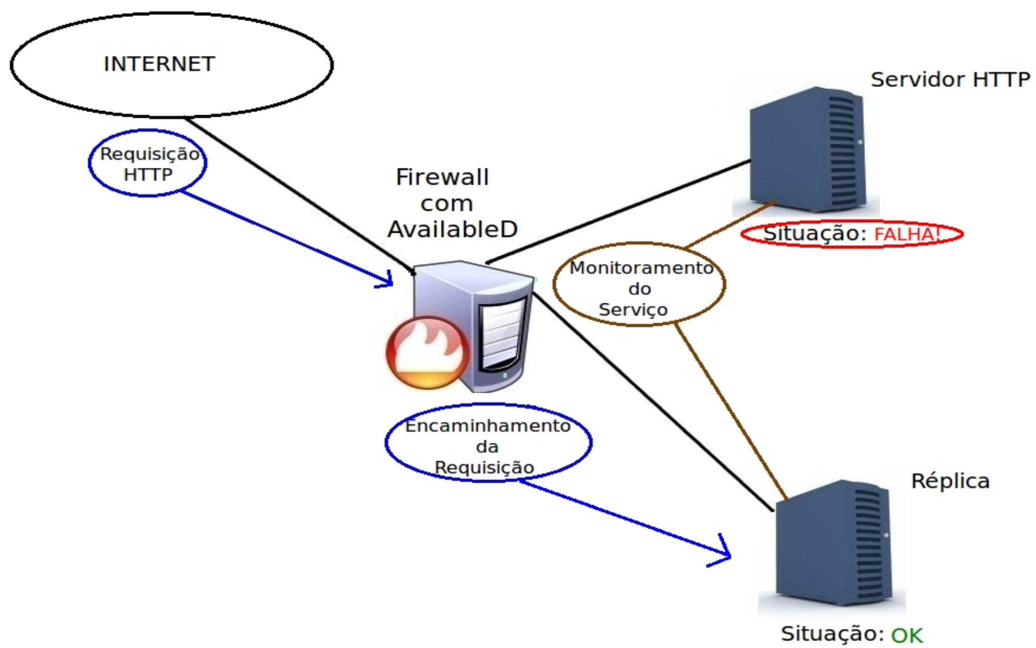


Figura 5: Funcionamento em ocasião de falha de um dos servidores.

1.5. Tecnologias e Requisitos de Sistema

O sistema tem como um de seus objetivos ser gratuito. Sendo assim, optou-se por distribuí-lo sob os termos da licença GNU GPLv3 (*General Public License version 3*). Desta maneira obtém-se também uma grande vantagem que é a possibilidade de membros e adeptos da comunidade do *software live* contribuírem no desenvolvimento do *AvailableD*.

Toda a tecnologia utilizada no desenvolvimento da ferramenta é livre. Abaixo está a lista de todas as tecnologias, ferramentas e projetos externos utilizados:

1. **Linguagem de programação C:** é a linguagem de base para o desenvolvimento da solução;
2. **Flex:** analisador léxico utilizado no módulo de configuração. Sítio do projeto: http://flex.sourceforge.net/;
3. **Mon:** ferramenta para monitoramento de serviços. Vários de seus monitores de serviço são adaptáveis e podem ser utilizados. Sítio do projeto: https://mon.wiki.kernel.org/;
4. **MSMTP:** é um cliente SMTP com suporte a vários métodos de autenticação, TLS, IPv6, dentre outras várias funcionalidades. Integrado ao *AvailableD* (evitando dependência) e utilizado como componente interno de notificação. Sítio do projeto: http://msmtp.sourceforge.net/;
5. **GnuTLS** e **OpenSSL:** ambas bibliotecas podem ser utilizadas pelo MSMTP para permitir utilização de TLS. Não são integradas ao *AvailableD* e não caracterizam uma dependência, mas sem nenhuma das duas, não é possível enviar email com utilização de segurança TLS. Sítios dos projetos: http://www.openssl.org/ e http://www.gnu.org/s/gnutls/;

A solução é implementada com enfoque em sistemas GNU/Linux, o que é justificável pela grande utilização deste sistema como *firewall*. Sendo assim, são previstas algumas dependências, como bibliotecas que só existem para este tipo de ambiente. Em um primeiro momento, a solução será específica para ambientes GNU/Linux. Não há previsão de compatibilização com outros sistemas.

No geral, os requisitos de sistema são os seguintes:

1. Sistema operacional GNU/Linux, com *firewall* instalado;
2. Servidor SMTP para envio de notificações (não obrigatório);

1.6. Documentação e Tratamento de Ajuda ao Usuário

A documentação, a nível de usuário, está presente no próprio arquivo de configuração do *AvailableD*. Tendo a simplicidade como princípio, o conjunto de conhecimentos necessários para utilizar a ferramenta não deve ser extenso, o que condicionou uma documentação bastante simples.

A nível de desenvolvedores, a documentação consiste neste documento, no documento de requisitos e a documentação de implementação constante nos códigos fontes no formato JAVADoc. Dessa forma, é possível gerar a documentação em HTML, PDF, dentre outros formatos com a utilização de ferramentas como *Doxygen* (http://doxygen.org).

A ajuda ao usuário, em tempo de execução do sistema, ocorrerá apenas durante o carregamento da configuração e na forma de mensagens emitidas. Como trata-se de um

daemon, sua execução ocorre em *background* e não é possível interação com o usuário da forma tradicional. A documentação do sistema deverá ser suficiente para sanar quaisquer eventuais dúvidas dos usuários.

2. Projeto Lógico e Arquitetural

2.1. Especificação Funcional

Ao executar o *AvailableD* , o mesmo deve ler seu arquivo de configuração, no qual deverá estar especificado:

a) Serviços a terem sua disponibilidade monitorada e suas respectivas informações. Exemplos de informações: endereço do servidor, porta de escuta, programa que verificará a disponibilidade do serviço, tempo de espera entre cada verificação, *timeout* de verificação, comandos de redirecionamento, etc. Deve ser possível informar a ordem de prioridades dos servidores de um mesmo serviço, ou seja, qual é o servidor primário, secundário, terciário, etc;
b) Informações referentes ao componente de notificação. Ser for um componente de envio de email, por exemplo, são necessárias informações do tipo: endereço do servidor, porta, usuário, senha, etc. Deve ser possível configurar para não enviar notificações.
c) Informações referentes a *log* de informações.

Ao carregar a configuração, se qualquer problema for identificado no arquivo, ou na própria configuração, o erro deve ser reportado ao usuário em forma de mensagem e a execução do programa deve ser interrompida. Todos os dados da configuração devem ser testados para verificar a validade/consistência dos mesmos.

Depois de ter sua configuração carregada, o sistema deve iniciar o monitoramento dos serviços para os quais foi configurado. Cada serviço monitorado em um servidor consistirá em um fluxo de execução independente (*thread*), o que permite que o monitoramento de servidores seja realizado em paralelo. Se ocorrer algum problema ao iniciar o monitoramento de qualquer serviço, o erro ocorrido deve ser exibido na forma de mensagem e o programa deve terminar sua execução.

O próximo passo é mudar o modo de execução para *background*, caracterizando um *daemon.* Também deve ser criado um arquivo no qual será gravada a identificação do processo no sistema operacional (PID). Este arquivo deve ser bloqueado com a finalidade de impedir que novas instâncias do processo sejam lançadas, o que não é permitido devido a condições de corrida que causariam.

No fluxo de monitoramento de cada servidor, serão realizadas as verificações para determinar a disponibilidade ou não dos serviços. Para tal, executa-se o programa responsável pela verificação do serviço (definido na configuração). Sempre que for identificado que um serviço está indisponível no servidor, deve ser adicionada uma regra de roteamento que redirecione os pacotes deste serviço, destinados ao servidor indisponível, para o servidor do mesmo serviço com maior prioridade para substituí-lo (definido na configuração) que esteja disponível.

Quando o serviço voltar a ficar disponível no servidor, as regras de redirecionamento relacionadas a ele deverão ser excluídas.

Sempre que houver alteração na situação dos servidores (disponível/indisponível) ou for realizada alguma alteração nas regras de roteamento, deve ser emitida notificação informando o ocorrido e deve ser feito *log* destas informações.

Ao terminar de realizar a verificação do serviço, a próxima verificação neste mesmo servidor deve ocorrer somente após um período de tempo especificado na configuração.

O programa somente irá terminar sua execução quando o usuário executar um comando de parada do sistema.

A figura 6 mostra o fluxograma geral de execução do *AvailableD* e a figura 7 o fluxo de execução do monitoramento em cada servidor.

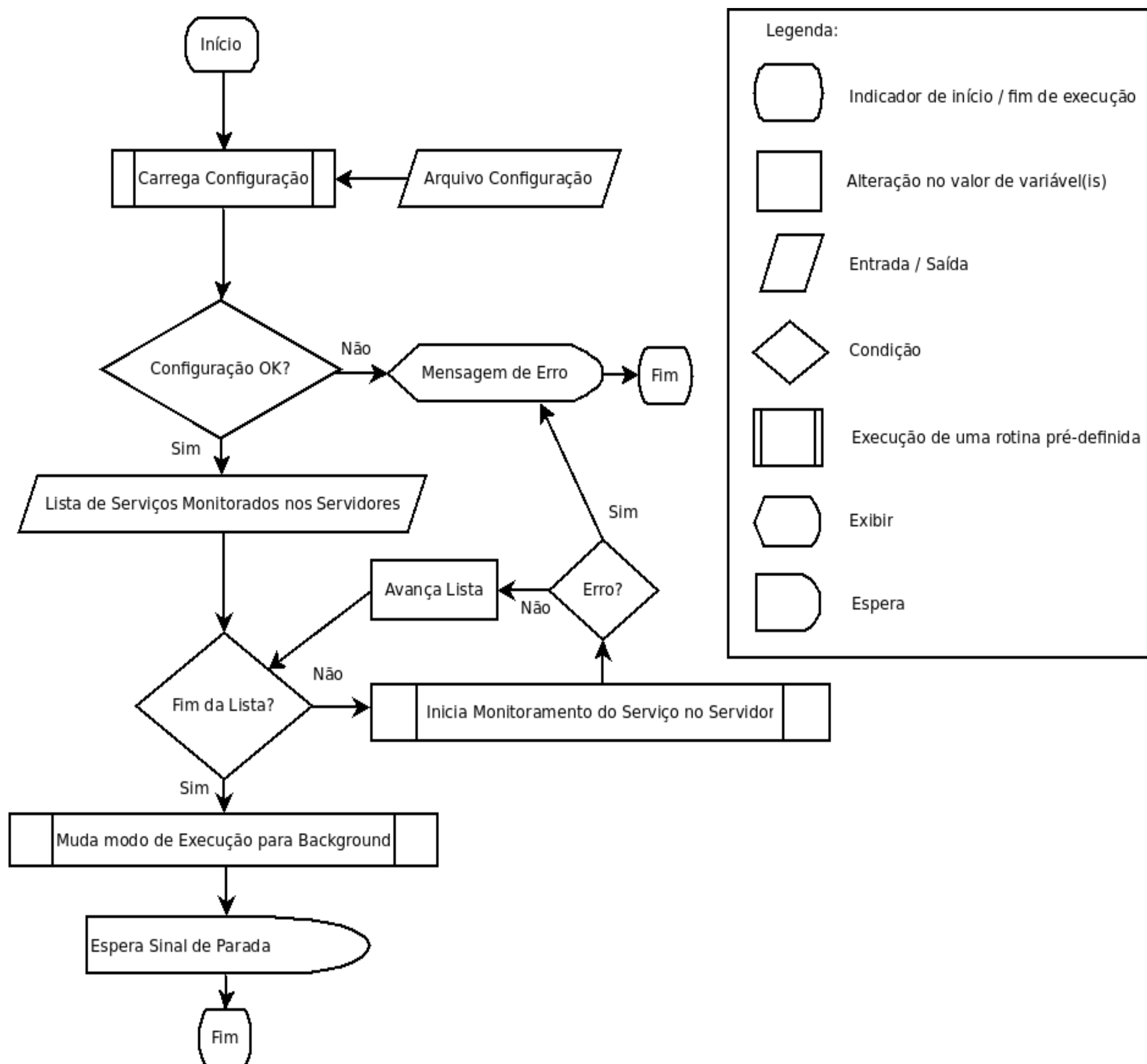


Figura 6: Fluxograma geral de execução do *AvailableD*.

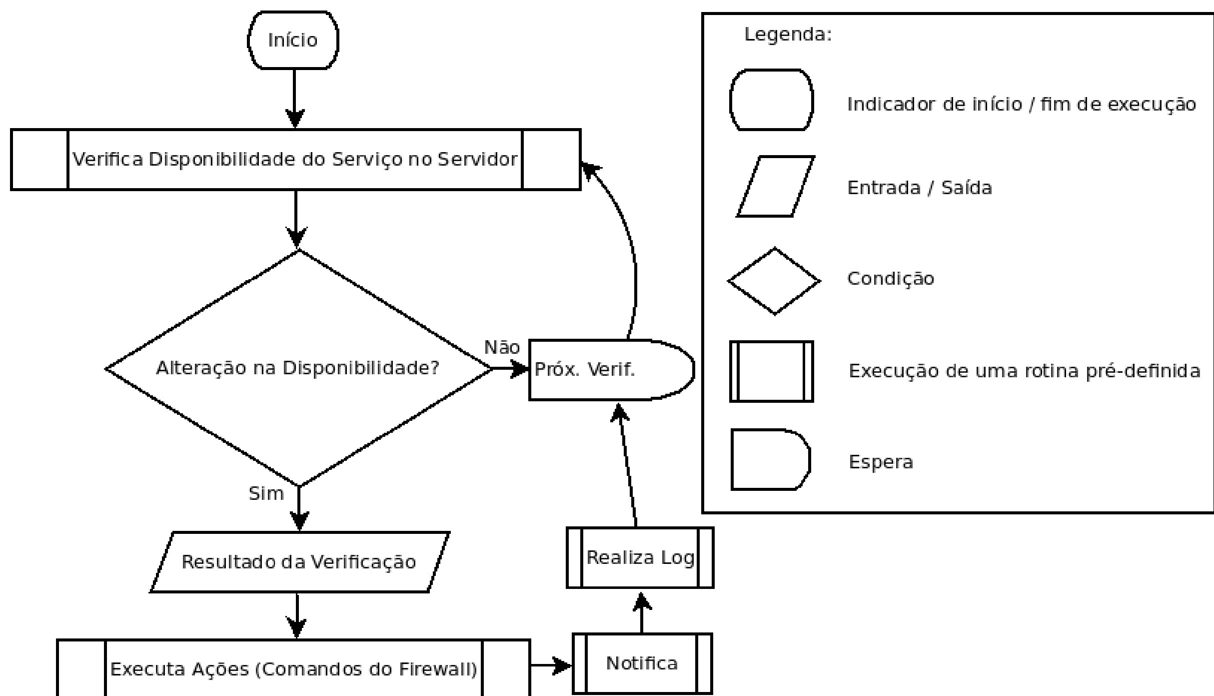


Figura 7: Fluxo de execução do monitoramento de serviços em cada servidor.

2.2. Arquitetura

Para satisfazer a especificação funcional e seus requisitos, a arquitetura definida para o *AvailableD* é composta dos seguintes módulos:

1. **Configuração:** responsável por ler, gerenciar e armazenar a configuração de todo o sistema. É o responsável por obter os parâmetros de configuração para os demais módulos;
2. **Monitoramento:** realiza as verificações de disponibilidade de serviços. Para isso, utiliza programas externos, que conectam-se aos servidores e testam o serviço. As verificações são executadas em paralelo, permitindo que vários serviços/servidores sejam monitorados simultaneamente. É responsável por determinar a situação de disponibilidade dos serviços;
3. **Executor de ações:** executa as ações configuradas de acordo com os resultados do monitoramento. As funcionalidades deste módulo, não podem ser executadas em paralelo entre *threads* de um mesmo serviço, a fim de evitar condições de corrida como, por exemplo, adição de redirecionamento para servidor que mudou seu estado para indisponível após o módulo ter recebido a informação que o mesmo estava disponível. Isso acarretaria na execução de redirecionamento para um servidor indisponível que já teve os redirecionamentos com destino a ele removidos e movidos para outros servidores. Então, o procedimento não será mais executado e o redirecionamento torna-se inútil. As ações consistem de comandos (*plugins*) do *firewall* para ativar ou remover os redirecionamentos de pacotes;
4. **Log:** envia mensagens recebidas para gerenciadores de *log*, os quais realizam o armazenamento das mensagens;

5. **Notificação:** envia notificações aos administradores da rede utilizando plugins. Para realizar notificações, podem ser utilizados programas externos. As notificações sempre que possível são executadas de forma paralela;

6. **Integração:** realiza a integração entre os demais módulos, possibilitando a comunicação entre eles. Responsável pela lógica de operação do *AvailableD*.

A figura 8 mostra a representação da arquitetura interna do *AvailableD*, levando em consideração. A figura 9 exibe também a estrutura e interação com os sistemas alvo das verificações. Na figura 10 está representada a estrutura e a interação com *plugins* e programas externos utilizados.

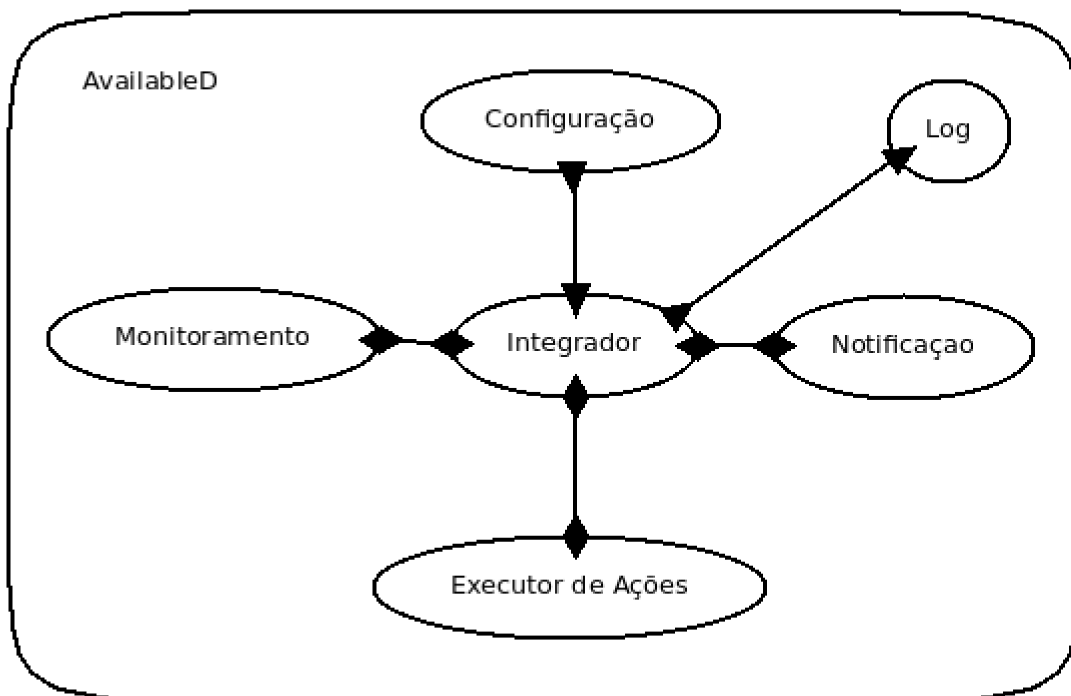


Figura 8: Arquitetura do *AvailableD.*

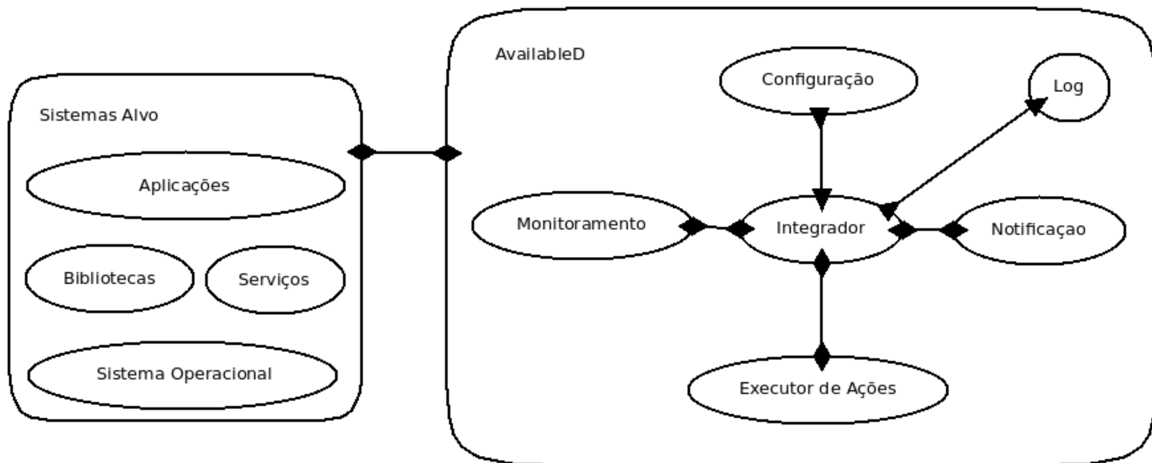


Figura 9: Arquitetura do *AvailableD* e dos sistemas alvo das verificações.

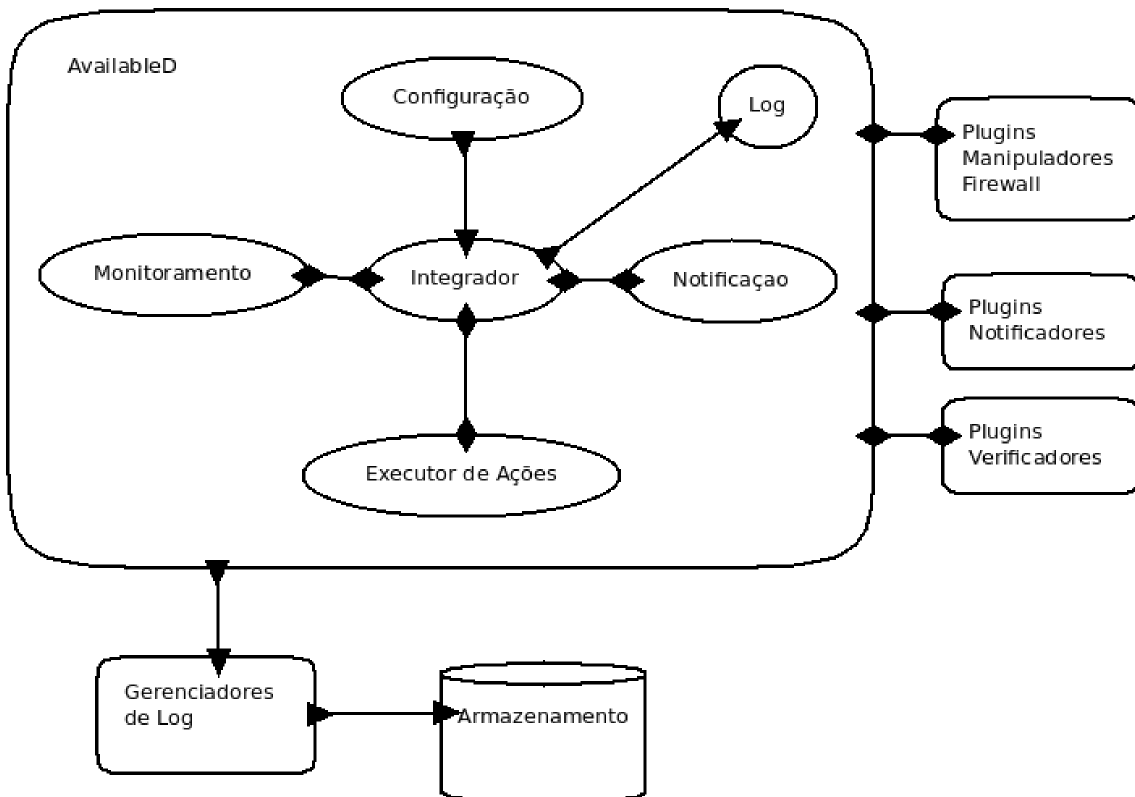


Figura 10: Arquitetura do *AvailableD* e sua interação com *plugins* e sistemas externos.

3. Projeto de Componentes, Dados e Interfaces

3.1. Configuração

A configuração do *AvailableD* fica toda armazenada em um arquivo texto. A justificativa para não utilizar ferramentas como banco de dados repousa na busca da simplicidade. Além do mais, a utilização de sistemas externos gera dependência que tem de ser resolvida pelo usuário.

O nome do arquivo de configuração é *availabled.conf* e fica localizado na raiz do diretório do *AvailableD* (por padrão */etc/availabled/*).

O arquivo é estruturado em 3 escopos:

1. **Global**: parâmetros declarados neste escopo constituem de padrões globais, valendo para todos os serviços/servidores;
2. **Serviço**: escopo onde as definições específicas de cada serviço são declaradas. Os parâmetros declarados neste escopo prevalecem sobre os declarados no escopo *global*;
3. **Servidor**: interno ao escopo serviço. Os parâmetros declarados neste escopo prevalecem sobre os declarados no escopo *serviço* e *global*.

A maior parte dos parâmetros de configuração são definidos para cada serviço presente em cada servidor. Isto significa que o escopo de declaração destes parâmetros, seria o escopo *servidor.* Para melhor estruturação da configuração e como uma forma de evitar repetição de informações comuns, é permitida a declaração de muitos destes parâmetros nos outros escopos, onde tornam-se definições de valor padrão para os parâmetros. Alguns parâmetros não admitem declaração em determinados escopos, porém, não geram um erro, somente são ignorados.

A sintaxe do arquivo consiste de 3 tipos de declarações:

1. **Parâmetros:** atribuição de valores aos parâmetros no formato: *parametro* = *valor1* [, valor2 [, valor3...]];. Nem todos os parâmetros são multi-valorados (aceitam mais de um valor);
2. **Serviços:** define informações a respeito de um serviço. Imediatamente muda o escopo de *global* para *serviço*. Formato da declaração: *service nome_servico* {}. O abre e fecha chaves serve para delimitar as declarações internas ao serviço;
3. **Servidores:** declarados sempre dentro dos serviços, indicam o início das informações pertinentes e específicas ao servidor. Imediatamente muda o escopo de *serviço* para *servidor*. Formato da declaração: server *identificacao_servidor* {}. Da mesma maneira que na declaração de serviços, o abre e fecha chaves serve para delimitar as declarações internas ao servidor. A identificação do servidor consiste em IP e, opcionalmente, a porta para acesso ao serviço.

A seguir, é apresentada a estrutura básica de um arquivo de configuração:

```
parametro1 = valor1;
parametro2 = valor2;
…
service name1 {

   parametro1 = valor1;
   …

   server ip1:porta1 {
       parametro1 = valor1;
       …
   }
   server ip2:porta2 {
       parametro1 = valor1;
       …
   }
   …

}
service name2 ...
```

Abaixo encontra-se a gramática que descreve a sintaxe do arquivo de configuração:

CONF → DEFGLOB
DEFGLOB → DEFPARAM | DEFSERVICE | DEFGLOBDUP
DEFGLOBDUP → DEFGLOB DEFGLOB
DEFPARAM → *param* = VALUELIST;
VALUELIST → *value* | value, VALUELIST
DEFSERVICE → service *name* {SERVICEINFO}
SERVICEINFO → DEFPARAM | DEFSERVER | SERVICEINFODUP
SERVICEINFODUP → SERVICEINFO SERVICEINFO
DEFSERVER → server *identification* {SERVERINFO}
SERVERINFO → DEFPARAM | DEFPARAM SERVERINFO

Definição dos Terminais da gramática:

1. *param*: nome do parâmetro.
   Caracteres aceitos: a-z, 0-9 e _.
   Condição: começa somente com letra, máximo de 25 caracteres;
2. *value*: valor para o parâmetro. Os valores podem ser representados entre aspas duplas ou não.
   Caracteres aceitos: No caso de não serem representados entre aspas, não adimitem a utilização dos seguintes caracteres: \n (nova linha), " (aspas duplas), " " (espaço em branco), "," (vírgula) e ";". Quando representado entre aspas, aceita todos os caracteres, porém o \n e o " tem que serem escapados com o \. Ex: \".
   Condição: não tem;
3. *name*: nome do serviço. Definido pelo usuário.

Caracteres aceitos: a-z, 0-9, _ e -.
    Condição: começa somente com letra, máximo de 30 caracteres.

4. *identification*: Identificação do servidor. Endereço IP encoberto ou não por “[]” e seguido, opcionalmente, do caracter “:” e da porta do serviço. Ex: [::1]:8080, 10.10.10.1:25, [192.168.0.1], [192.168.0.1]:5432.
    Caracteres aceitos: 0-9, ., [,] e :.
    Condição: IP e porta válidos;
5. *service* e *server*: palavras reservadas para indicar início da definição de um serviço e de um servidor, respectivamente;
6. “{“ e “}”: indicam início e termino de uma nova seção/escopo;
7. “=”: indica atribuição;
8. “,”: delimitador de valores;
9. “;”: delimitador de atribuições;
10. Na gramática não está representado como é realizado o comentário por motivos de maior legibilidade, mas é possível escrever comentários. O caractere que indica início de um comentário é o “#”, desde que informado separado da definição de valores (caso em que é considerado parte do valor). Após este caractere, o resto da linha toda será ignorada (até o \n);
11. Os delimitadores de componentes da gramática são: \n (nova linha), “ “ (espaço em branco) e \t (tabulação).

Como falado anteriormente, um mesmo parâmetro pode ser declarado em vários escopos, o que serve para definição de valores padrão. Há regras de prevalecência entre os diferentes escopos, o que torna funcional a utilização de valores padrões. Para entender a prevalecência entre os escopos, assuma a seguinte configuração:

```
parametro = valor_global;

service ficticio {

  parametro = valor_servico;

  server  10.10.10.1 {
    parametro = valor_servidor;
  }

  Server 10.10.10.2 {
  }

}

service abstrato {
  server 10.10.10.3 {
  }
}
```

Assumindo que *parametro* aceita declaração em qualquer escopo, para o servidor 10.10.10.1, seu valor será "valor_servidor", pois o escopo *servidor* possui maior prevalecência que os demais. No servidor 10.10.10.2, o valor de *parametro* será "valor_servico", visto que o mesmo não foi informado no escopo *servidor* mas foi informado no escopo *serviço*. Por fim, no servidor 10.10.10.3, *parametro* vale "valor_global", pois não há nenhuma outra declaração de maior prevalecência no serviço *abstrato*. Para parâmetros multi-valorados, o funcionamento é o mesmo, ou seja, há a sobrescrita de valores em vez da adição.

Outro aspecto importante do arquivo de configuração é que o lugar onde os parâmetros são declarados dentro de seu escopo não influencia. Ainda utilizando a configuração acima, se colocássemos a primeira linha (parametro = valor_global;) no final da configuração, ou seja, se ela passasse a ser a última linha, após a declaração do serviço *abstrato*, mesmo assim o servidor 10.10.10.3 possuiria *parametro* valendo "valor_global". Por questões de legibilidade, não recomenda-se declarações após o término de um outro escopo, ou seja, as declarações de parâmetros devem ficar imediatamente no início do escopo ao qual pertencem.

Há alguns detalhes necessários entender quanto a declaração repetida dos parâmetros dentro do mesmo escopo:

1. Para parâmetros que não são multi-valorados, a última declaração dentro do mesmo escopo é a que vale. Se um valor não é multi-valorado, mas for declarado como tal, somente o último valor listado será utilizado. Ex: parametro_univalorado = 1, 2, 3; resultará em parametro_univalorado valer 3;
2. Para parâmetros multi-valorados, cada declaração equivale a adição de valor para o parâmetro. Declarar este tipo de parâmetro várias vezes no mesmo escopo equivale a declará-lo uma única vez e separar seus valores por ",".

Abaixo estão listados todos os parâmetros de configuração com seus respectivos significados e informações relevantes. Para melhor organização, estão agrupados de acordo com o módulos que os utilizam:

1. **Monitoramento:**
   a) ***check_path:*** diretório(s) no qual deverão ser procurados os programas de verficação dos serviços. Os caminhos especificados devem ser absolutos. Esta variável é adicionada à variável de ambiente PATH, portanto, para adicionar mais de um diretório, os caminhos devem ser separados por ":". Observe que embora aceite mais de um diretório, este parâmetro não é multi-valorado na visão do sistema.
   Valores aceitos: diretórios válidos separados por ":". Ex: /tmp, /tmp:/home.
   Multi-valorado: Não.
   Escopo: global, serviço e servidor.
   Obrigatório: Não.
   b) ***check_command:*** comando a ser executado para verificar a disponibilidade do serviço monitorado. Este comando aceita o uso das seguintes variáveis:
      i. $SERVER – endereço IP do servidor. Será substituída pelo endereço IP informado na identificação do servidor.
      ii. $PORT – porta para se conectar no serviço. Será substituída pela porta informada na identificação do servidor.

Valores aceitos: comandos válidos. Ex: “http.monitor --path \”index.html\" --match-regex \".*TESTE[<>]{1}\" --port $PORT $SERVER";
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Sim.

c) ***check_timeout***: configuração do tempo necessário para determinar que um serviço não está respondendo à verificação e considerá-lo indisponível. Se durante a verificação de um serviço, não for obtido um retorno em *check_timeout* tempo. a verificação é interrompida e o serviço é considerado indisponível.
Valores aceitos: números inteiros positivos acompanhados ou não das letras maiúsculas ou minúsculas *s* (segundos), *m* (minutos), *h* (horas) e *d* (dias). Se nenhuma letra for informada, será assumida a unidade segundos. Ex: 100, 100s, 1m40s, 2h10s, 4d2h.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Sim.

d) ***wait_time***: tempo entre as execução das verificações do serviço. Após verficar a situação de um serviço em determinado servidor, o sistema irá esperar *wait_time* tempo para realizar a próxima verificação no mesmo servidor.
Valores aceitos: segue as mesmas especificações de *check_timeout*.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Sim.

2. Executor de Ações:

a) ***action_commands***: comandos que serão executados quando o serviço for considerado indisponível. É neste parâmetro que vão os comandos do *firewall* para adicionar redirecionamento de pacotes. Os comandos são executados na ordem em que são informados. Estes comandos aceitam o uso das seguintes variáveis:

   i. $BROKEN_SERVER – servidor indisponível. Será substituída pelo endereço IP informado na identificação do servidor.
   ii. $BROKEN_PORT – porta do servidor indisponível. Será substituída pela porta informada na identificação do servidor.
   iii. $ACTIVE_SERVER – servidor escolhido para receber os pacotes do servidor indisponível. Será substituída pelo endereço IP informado na identificação do servidor escolhido.
   iv. $ACTIVE_PORT – porta do servidor escolhido para receber os pacotes do servidor indisponível. Será substituída pela porta informada na identificação do servidor escolhido.

Valores aceitos: comandos válidos. Ex: “iptables -t nat -A PREROUTING -p tcp -d $BROKEN_SERVER --dport $BROKEN_PORT -j DNAT –to $ACTIVE_SERVER:$ACTIVE_PORT”.
Multi-valorado: Sim.
Escopo: global, serviço e servidor.
Obrigatório: Não. O usuário pode desejar somente ser notificado, por exemplo.

b) ***undo_action_commands***: comandos que serão executados quando o serviço for considerado novamente disponível. É neste parâmetro que vão os comandos do *firewall* para remover o redirecionamento de pacotes realizado pelos *action_commands*. Os comandos são executados na ordem em que são informados. Este comando possui as mesmas variáveis e a mesma definição para os valores aceitos de *action_commands.* Obs: as variáveis $BROKEN referem-se ao servidor que **estava** indisponível enquanto que as variáveis $ACTIVE referem-se ao servidor para o qual o redirecionamento havia sido realizado. Ex: "iptables -t nat -D PREROUTING -p tcp -d $BROKEN_SERVER --dport $BROKEN_PORT -j DNAT –to $ACTIVE_SERVER:$ACTIVE_PORT".
Multi-valorado: Sim.
Escopo: global, serviço e servidor.
Obrigatório: Não.

c) ***id***: número de identificação única para cada servidor de um serviço. Em serviços diferentes, os servidores podem possuir IDs iguais.
Valores aceitos: números inteiros positivos.
Multi-valorado: Não.
Escopo: servidor.
Obrigatório: Sim.

d) ***redirect_to***: lista de servidores do mesmo serviço para os quais é aceitável o redirecionamento de pacotes. A ordem na qual os servidores forem informados representa a prioridade dos mesmo em atender as requisições do servidor falho. Este parâmetro recebe os IDs dos servidores.
Valores aceitos: inteiros positivos não nulos que representem IDs válidos de servidores do mesmo serviço.
Multi-valorado: Sim.
Escopo: servidor.
Obrigatório: Não. O usuário pode desejar não redirecionar os pacotes de um servidor indisponível.

3. **Log:**

a) ***log_file***: endereço para o arquivo de log. O caminho deve ser absoluto.
Valores aceitos: endereço de arquivos com permissão de escrita. Ex: /var/log/availabled.log.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Sim.

b) ***log_facilities***: gerenciadores de *log* utilizado.
Valores aceitos: para a primeira versão implementada, o único valor aceito é "file".

4. **Notificação:** os parâmetros de notificação contém as informações necessárias pelas aplicações responsáveis de emitir notificação. O *AvailableD* possui um cliente SMTP (o MSMTP) integrado. Sendo assim, os parâmetros que iniciam com a palavra "mail" referem-se a parâmetros para envio de email utilizando este cliente. É permitido utilização de qualquer cliente externo, desde que sua interface suporte recebimento de argumentos pela linha de comando.

a) ***notification_commands***: comandos (utilizando programas externos) para enviar notificação. Este parâmetro aceita o uso das seguintes variáveis:
$MESSAGE – Mensagem de notificação gerada pelo *AvailableD*.
Valores aceitos: comandos válidos. Ex: "smsscript -t +555599999999 -m \"$MESSAGE\"".
Multi-valorado: Sim.
Escopo: global, serviço e servidor.
Obrigatório: Não.
mail_notification: determina se o *AvailableD* irá ou não enviar notificações por email utilizando seu cliente SMTP interno.
Valores aceitos: *true*, *false*, *yes* e *no*.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não. Se não informado, assume o valor não.

b) ***mail_user***: nome de usuário utilizado na autenticação para o envio de email. Também utilizado como remetente do email.
Valores aceitos: endereços de email válidos. Ex: somebody@domain.com.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Somente se *mail_information* for sim.

c) ***mail_pass*:** senha do para o usuário utilizado na autenticação para o envio de email.
Valores aceitos: strings. Ex: my#passs
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

d) ***mail_to***: lista de emails que receberão as notificações.
Valores aceitos: endereços de email válidos. Ex: netadmin@domain.com.
Multi-valorado: Sim.
Escopo: global, serviço e servidor.
Obrigatório: Somente se *mail_information* for sim.

e) ***mail_server***: endereço do servidor de email utilizado para enviar os emails.
Valores aceitos: endereço IP ou nome de domínio do servidor. Ex: 10.10.10.1, smtp.mydomain.com.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Somente se *mail_information* for sim.

f) ***mail_port***: número da porta para comunicação com o servidor de email.
Valores aceitos: 1-65535. Valor padrão do protocolo SMTP: 25.
Multi-valorado: Não.
Escopo: global, serviço e servidor.

g) ***mail_subject***: assunto dos emails enviados.
Valores aceitos: strings. Ex: "[AvailableD] Informações sobre alteração da disponibilidade de serviço de rede".
Multi-valorado: Não.
Escopo: global, serviço e servidor.

Obrigatório: Somente se *mail_information* for sim.

h) ***mail_auth_mech***: método de autenticação a ser utilizado para envio de email, se houver autenticação.
Valores aceitos: CRAM-MD5, PLAIN, EXTERNAL, LOGIN, DIGEST-MD5, SCRAM-SHA-1, GSSAPI e NTLM. Nem todos estes métodos são suportados se o componente de email (msmtp) não for instalado com suporte a biblioteca GNU SASL.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

i) ***mail_helo_domain***: domínio utilizado para se identificar para o servidor de email. Esta opção é útil principalmente quando o servidor SMTP faz validações do domínio informado. Geralmente esta opção será o domínio do usuário de email utilizado para o envio.
Valores aceitos: domínios, IPs. Ex: mysmtpdomain.com.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

j) ***mail_ntlm_domain***: domínio utilizado para autenticação do tipo NTLM. Esta opção somente é útil quando se a autenticação for NTLM.
Valores aceitos: domínios, Ips. Ex: myorg.hr
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

k) ***mail_use_tls***: determina se usa ou não tls na comunicação com o servidor de email.
Valores aceitos: *true*, *false*, *yes* e *no*.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não. Se não informado, assume não como valor.

l) ***mail_tls_cert_file***: caminho para o arquivo de certificado TLS, se utilizado.
Valores aceitos: arquivos com permissão de leitura. Ex: /etc/tls/cert.pem.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

m) ***mail_tls_key_file***: caminho para o arquivo com a chave TLS.
Valores aceitos: arquivos com permissão de leitura. Ex: /etc/tls/private.key.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

n) ***mail_tls_revoke_list_file***: caminho para o arquivo com a lista de revogação.
Valores aceitos: arquivos com permissão de leitura. Ex: /etc/tls/revoke.crl.
Multi-valorado: Não.
Escopo: global, serviço e servidor.
Obrigatório: Não.

o) ***mail_tls_trust_file***: este parâmetro ativa uma rigorosa verificação do certificado do servidor. Caminho para o arquivo com a lista Autoridades de Certificação de confiança.
   Valores aceitos: arquivos com permissão de leitura. Ex: /etc/tls/ca-certificates.crt.
   Multi-valorado: Não.
   Escopo: global, serviço e servidor.
   Obrigatório: Não.
p) ***mail_check_tls_cert***: determina se verifica ou não o certificado TLS do servidor.
   Valores aceitos: *true*, *false*, *yes* e *no*.
   Multi-valorado: Não.
   Escopo: global, serviço e servidor.
   Obrigatório: Não. Se não informado, assume não como valor.
q) ***mail_starttls***: determina se deve ou não utilizar o comando SMTP STARTTLS. Quando estiver definido para utilizar TLS e este parâmetro for definido como não, o TLS será iniciado logo ao conectar com o servidor.
   Valores aceitos: *true*, *false*, *yes* e *no*.
   Multi-valorado: Não.
   Escopo: global, serviço e servidor.
   Obrigatório: Não. Se não informado, assume não como valor.

Abaixo, temos um exemplo de arquivo de configuração para monitorar o serviço HTTP em 3 servidores, utilizando notificação por email:

```
# Configuração global de email
mail_notification = yes;
mail_user = myuser@mydomain.com;
mail_pass = mysecret;
mail_to = root@mydomain.com, support@mydomain.com;
mail_server = smtp.mydomain.com;
mail_port = 25;

# Informação global de monitoramento dos serviços
check_path = /etc/availabled/checkers
wait_time = 3s;
check_timeout = 5s;

# Informação global de log
log_file = /var/log/availabled.log;

# Configurações globais do executor de ações
action_commands = "iptables -t nat -A PREROUTING -p tcp  -d $BROKEN_SERVER \
                   --dport $BROKEN_PORT -j DNAT \
                   --to $ACTIVE_SERVER:$ACTIVE_PORT";

undo_action_commands = "iptables -t nat -D PREROUTING -p tcp \
                        -d $BROKEN_SERVER --dport $BROKEN_PORT \
                        -j DNAT –to  $ACTIVE_SERVER:$ACTIVE_PORT";

# Definição do serviço HTTP
service http {

 # Informação padrão do serviço
 check_command = "http.monitor --path \"index.html\"
                  --match-regex \".*TESTE[<>]{1}\" \
                  --port $PORT $SERVER";
 # Servidores
 server 10.10.10.1:80 {
   id = 1;
   redirect_to = 2, 3;
 }
 server 10.10.10.2:80 {
   id = 2;
   redirect_to = 1, 3;
 }
 server 10.10.10.3:80 {
   id = 3;
   redirect_to = 1, 2;
 }

}
```

Funcionamento do componente:

1) Abre o arquivo de configuração. Se não conseguir, vai par o passo 7;
2) Se atingiu o final do arquivo, vai para o passo 6, senão, identifica lexema e vai para o passo 3;

3) Verifica a conformidade da sintaxe. Se não está de acordo, vai para o passo 7, senão vai para o passo 4.
4) Verifica a validade do valor do lexema. Se não for válido, vai para o passo 7. Se o lexema é proveniente do símbolo da gramática que representa o valor de um parâmetro, vai para o passo 5, senão, vai para o passo 2;
5) Se o valor for válido para o parâmetro, seta variável de configuração com o valor do lexema e vai para o passo 2, senão vai para o passo 7;
6) Verifica se a configuração é válida (todas variáveis de configuração necessárias foram setadas com valores aceitos). Se é válida, vai para o passo 8 senão, vai para o passo 7;
7) Emite mensagem de erro e fecha o arquivo, se estiver aberto. Vai para o passo 8;
8) Termina a execução.

A Figura 11 mostra o fluxograma de execução do componente de configuração.

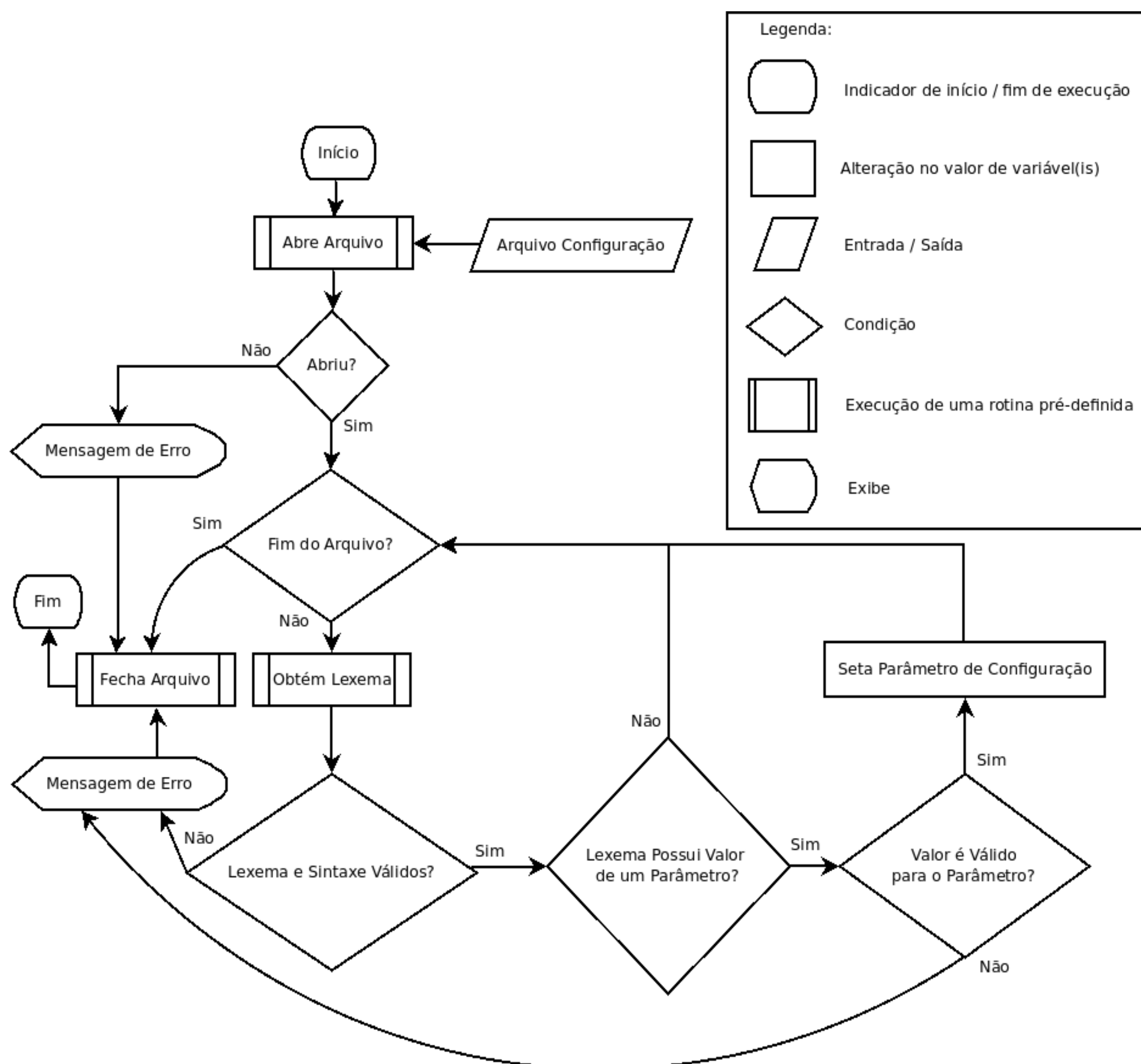


Figura 11: Fluxograma de execução do componente de configuração.

Tecnologias para Implementação:

Para implementar o módulo de configuração será utilizado o analisador léxico *flex.* Não será utilizado o *bison* (analisador sintático), pois não há necessidade devido a simplicidade da gramática. Para facilitar a correta identificação dos tokens, deverão ser utilizadas as condições de início (*start conditions*) disponibilizadas no *flex*. Através desta funcionalidade, é possível manter ativos somente os padrões esperados, o que retira a ambiguidade de lexemas e facilita a identificação de erros de sintaxe.

Estruturas de dados:

```
/* Estrutura de lista auxiliar */
able_list_t {

  void* opData; /*Dado */
  struct able_list_t* opPrev; /* Próximo nó da lista */
  struct able_list_t* opNext; /* Nó anterior da lista*/

};

/* Informações do serviço, servidor, monitoramento e notificação */
able_svcinfo_t {

  char* sIP; /* Endereço IP do servidor */
  unsigned int iID; /* ID do servidor no serviço */
  int iPort; /* Porta do serviço */
  long int iCheckTimeout; /* Timeout para rotina de verificação em segundos */
  long int iWaitTime; /* Tempo entre cada verificação do serviço em segundos */
  char* sCheckPath; /* Caminho onde comandos de verificação devem ser procurados */
  char* sCheckCommand; /* Comando de verficação */
  char* sLogFile; /* Caminho para o arquivo de log */
  bool bMailNotification; /* Terá notificação por email? */
  able_mail_t* opMailConf; /* Estrutura definida no módulo notificação */
  able_list_t* opRedirectTo; /* Lista de int com os IDs dos servidores para redirecão */
  able_list_t* opActionCommands; /* Lista de char* com comandos de redir. */
  able_list_t* opUndoActionCommands; /* Lista de char* com cmds pra rem. redir. */
  able_list_t* opNotificationCommands; /* Lista de char* com cmds de notificação */

} able_svcinfo_t;

/* Serviço com sua lista de servidores/informações */
typedef struct able_service_t {

  char* sName; /* Nome do serviço */
  able_list_t* opInfo; /* Lista de servidores e informações (lista de able_svcinfo_t*) */

} able_service_t;
```

Interfaces:

a) Entrada:
   i. const char* – caminho do arquivo de configuração;

b) Saída:
   i. able_list_t*(able_service_t*) – lista com todos os serviços configurados para monitoramento;
   ii. char** – irá receber mensagens de erro na posição 0.

3.2. Log

A informação de log do *AvailableD* é toda armazenada por gerenciadores de *log*. Para um primeiro protótipo, será implementado um gerenciador de *log* para em um arquivo definido na configuração. O arquivo deve seguir o formato CSV. Sua estrutura é a seguinte:

1) Data no formato: AAAA-MM-DD hh:mm:ss
2) Tipo de mensagem: Valores possíveis: ERROR, NOTICE e WARNING.
3) Mensagem: texto da mensagem.

Os campos do arquivo são separados por "," (CSV), o que facilita uma futura análise automatizada do arquivo.

Funcionamento do gerenciador de log para escrita em arquivo:

1) Tenta abrir o arquivo de *log*. Se conseguir, vai para o passo 2, senão, para o passo 6;
2) Obtém a data atual do sistema e vai para o passo 3;
3) Processa mensagem de *log* para que se adéque ao padrão CSV e vai para o passo 4;
4) Gera mensagem completa para ser escrita no arquivo e vai para o passo 5;
5) Adiciona a mensagem no arquivo e vai para o passo 6.
6) Termina a execução;

A figura 12 mostra o fluxograma do gerenciador de *log* para escrita em arquivo.

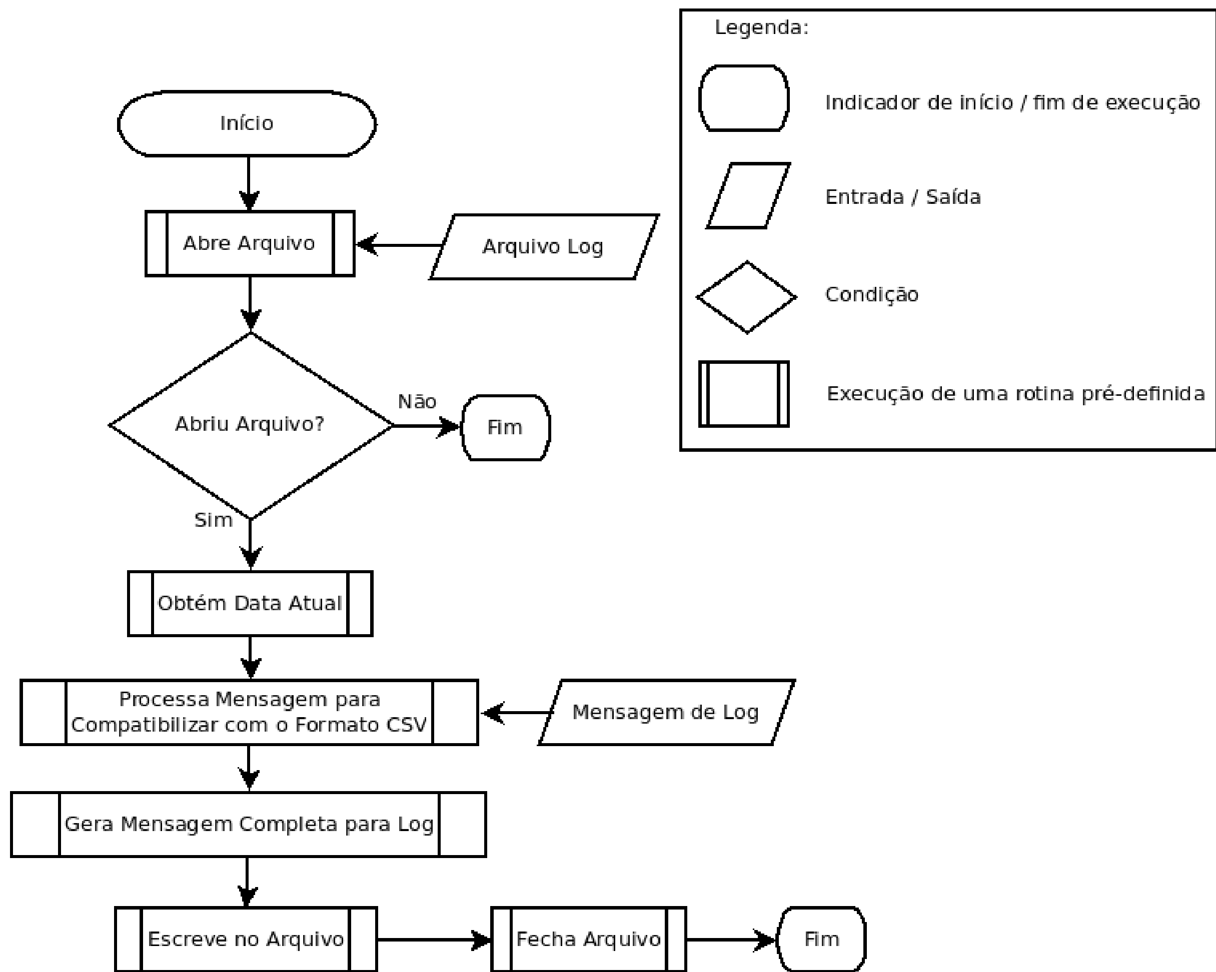


Figura 12: Fluxograma de execução do gerenciador de log para escrita em arquivo.

Tecnologias para Implementação:
Nenhuma tecnologia a ser especificada.

Estruturas de dados:

```
/* Mensagem de log */
able_logmsg_t {

  char sType[12]; /* Tipo de mensagem de log */
  char* sMessage; /* Mensagem de log */

};
```

Interfaces:

a) Entrada:
   i. able_logmsg_t* – mensagem para log;
   ii. const char* – caminho do arquivo de log.

b) Saída:
   i. bool – sucesso: true, falha: false.

3.3. Notificação

O componente de notificações é responsável por enviar mensagens detalhando eventos de grande relevância ocorridos na rede, como por exemplo, redirecionamentos realizados.

As mensagens são estruturadas de forma clara e objetiva, provendo as informações necessárias para que os problemas diagnosticados na rede sejam rapidamente resolvidos.

Este módulo permite a utilização de quaisquer programas/componentes de notificação externos, desde que a interface de linha de comando dos mesmos seja suficiente para o envio das notificações. Caso não sejam, ainda pode haver a possibilidade de ser implementado um programa/script que faça o meio de campo e possibilite a utilização do mesmo. A forma de passar a mensagem gerada pelo *AvailableD* para os programas é através da variável $MESSAGE que é interpretada quando especificada nos argumentos dos notificadores e substituída pelo texto da mensagem.

Para desenvolvimento de componentes de notificação, alguns detalhes devem ser observados:

1. **Valores de Retorno**: um notificador deve retornar 0 somente quando obteve sucesso em realizar a notificação. Em caso de erros ao notificar, qualquer outro valor deve ser retornado. Os valores de retorno 126 e 127 são reservados para identificar que sequer houve tentativa de realizar a notificação. Uma situação em que estes valor devem ser usados é durante a análise dos parâmetros passados. Se estiverem incompletos ou incoerentes, a notificação não poderá ser realizada, mas isso não significa que o componente falhou tentando realizar a notificação, o que poderia caracterizar algum problema no mesmo.
2. **Recursos Utilizados**: por questões de segurança, os notificadores devem ser cautelosamente desenvolvidos para não utilizarem recursos que possam consistir em brechas de segurança, principalmente porque poderão estar sendo executados com as credenciais de usuários privilegiados do sistema.
3. **Mensagens de Erro**: mensagens somente são utilizadas pelo *AvailableD* quando o notificador retornar algum código diferente de 0. Portanto, somente há necessidade de serem emitidas para informar problemas identificados. A maneira de passar estas mensagens para o *AvailableD* é imprimindo-as na saída padrão. Devem ser evitados caracteres especias.

Com a finalidade de prover um mecanismo de notificação próprio, possibilitando envio de mensagens sem a necessidade de programas externos, o *AvailableD* possui integrado um componente para envio de emails.

Funcionamento do componente

1) Obtém a lista de notificadores (comandos) externos a serem utilizados. Vai para o passo 2;
2) A lista de comandos de notificação está vazia? Se sim, vai para o passo 6, senão, obtém comando e vai para o passo 3;
3) Substitui variáveis pelos seus valores no comando de notificação. Vai para o passo 4;
4) Executa comando e vai para o passo 5;

5) Avança lista de comandos e vai para o passo 2;
6) Verifica se há algum componente interno configurado para ser utilizado. Se sim, vai para o passo 7, senão, para o passo 8;
7) Executa os componentes internos configurados para realizar notificação. Vai para o passo 8;
8) Termina execução.

Funcionamento do componente interno para envio de email:

1) Tenta conectar-se ao servidor. Se conseguir, vai para o passo 2, senão, vai para o passo 7;
2) Se configurado para usar TLS e não utilizar o comando STARTTLS, inicia automaticamente o TLS. Se ocorrer algum problema iniciando o TLS, vai para o passo 7, senão, para o passo 3;
3) Obtém informações do servidor, se possível (comando EHLO do protocolo SMTP). Caso algum recurso que não é suportado pelo servidor esteja configurado para ser utilizado, vai para o passo 7, senão, vai para o passo 4;
4) Se configurado para usar TLS e o comando STARTTLS, executa o comando e inicia o TLS. Se ocorrer algum problema iniciando o TLS, vai para o passo 7, senão, para o passo 5;
5) Se necessário autenticar, efetua o processo de autenticação utilizando os dados da configuração. Se não for possível autenticar, vai para o passo 7, senão, para o passo 6.
6) Executa o protocolo SMTP, realizando o envio do email. Vai para o passo 7.
7) Termina a execução.

A figura 13 mostra o fluxograma do componente notificação utilizando programas exeternos, e a figura 14 o fluxograma do componente interno para envio de email.

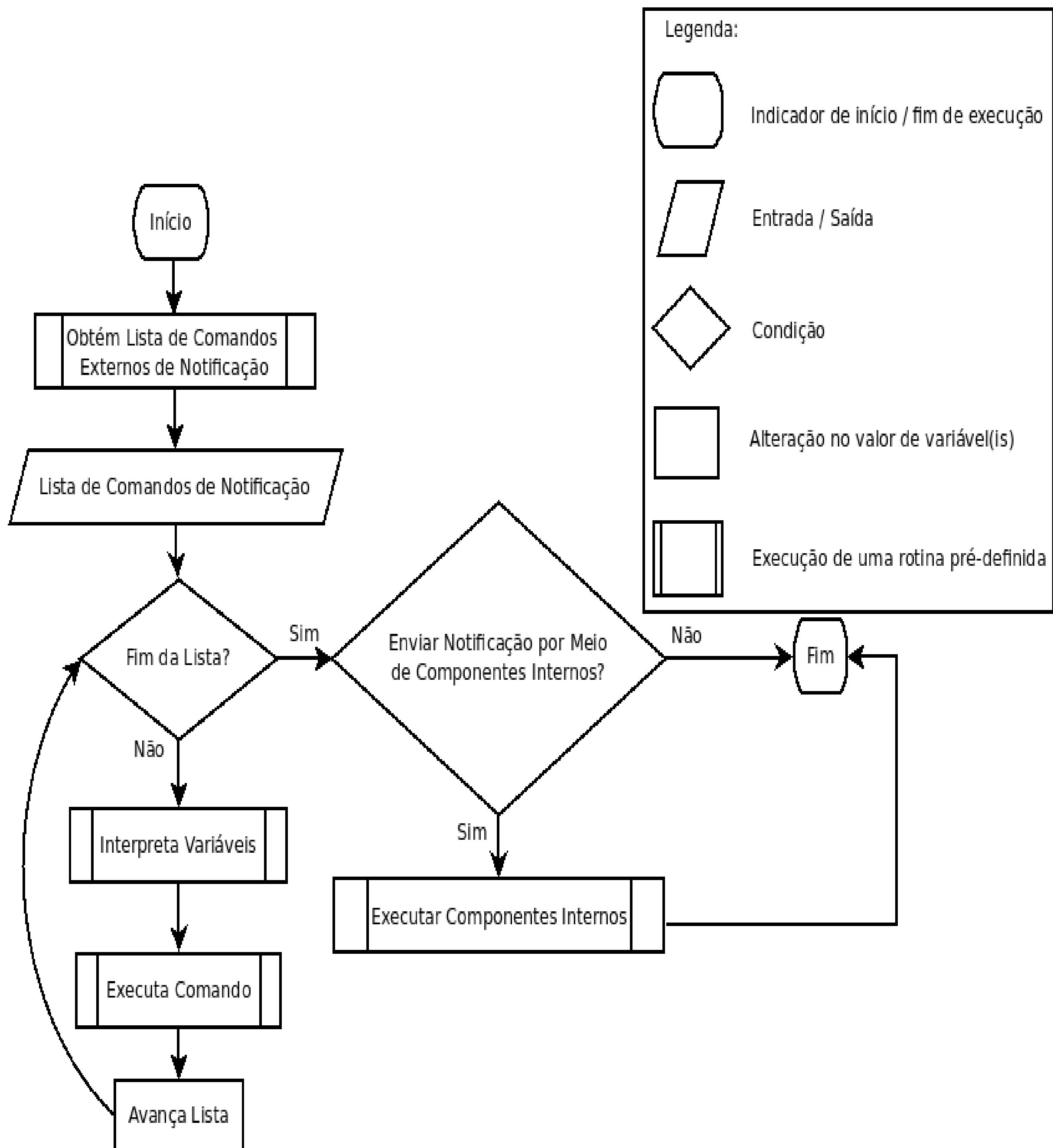


Figura 13: fluxograma do componente notificação utilizando programas externos.

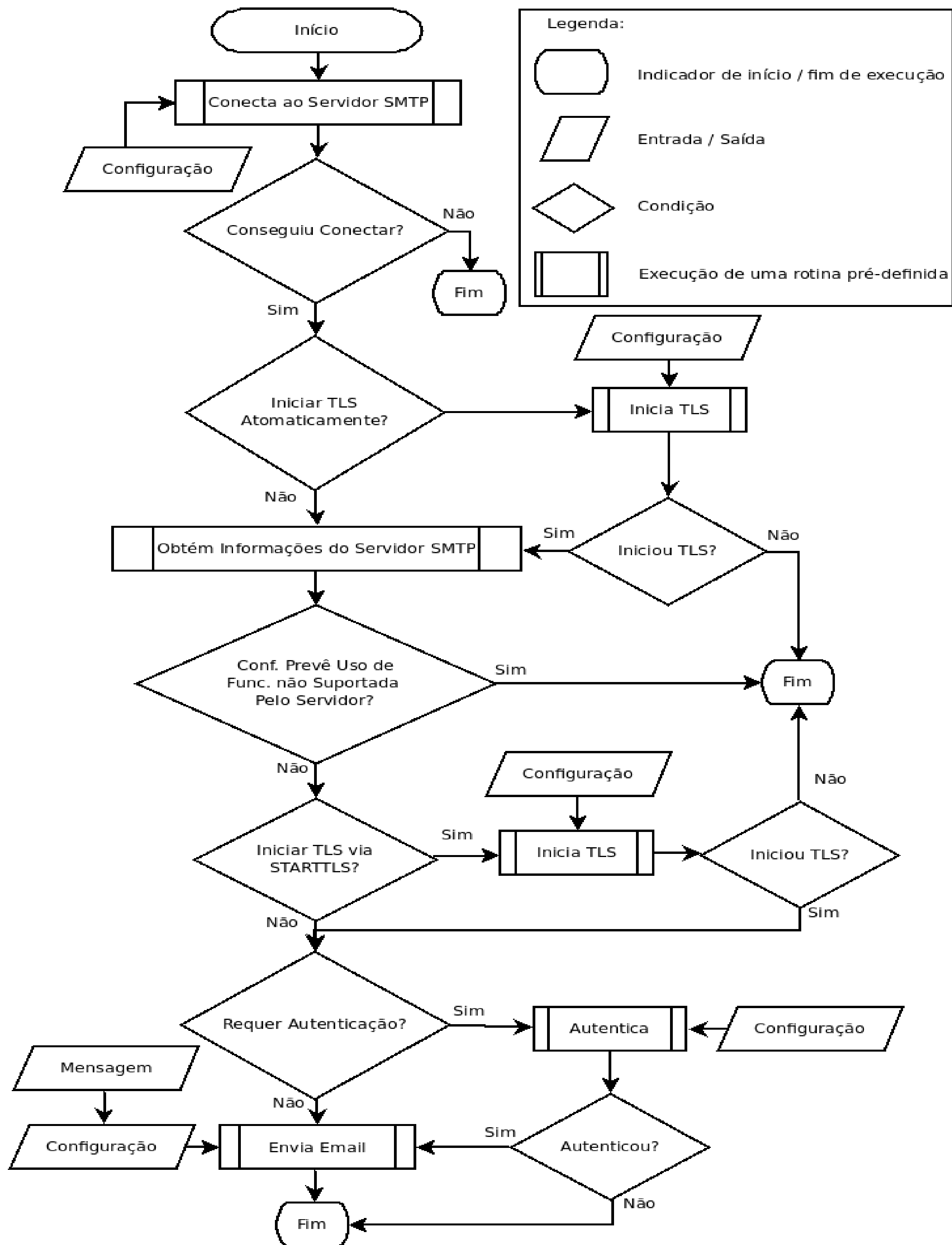


Figura 14: Fluxograma de execução do componente interno para envio de email.

Tecnologias para Implementação

Para implementar o componente interno de notificação por email, utilza-se a biblioteca SMTP do cliente MSMTP. A biblioteca é bastante simples e completa ao mesmo tempo.

Estruturas de dados:

```
/* Estrutura utilizada pelo componente interno para envio de email e pelo componente
de configuração */
able_mail_t {

  char* sUser; /* Usuário utilizado como remetente e na autenticação */
  char* sPass; /* Senha para o usuário. Utilizada na autenticação. */
  able_list_t* opTo; /* List de char* com os emails dos destinatários */
  char* sServer; /* Servidor de email. Endereço IP ou nome do servidor */
  int iPort; /* Porta para conexão com o servidor. */
  char* sSubject; /* Assunto do email */
  char* sData; /* Corpo do email */
  char* sHeloDomain; /* Domínio de identificação do cliente no cmd HELO */
  char* sNtlDomain; /* Domínio NTLM utilizado a autenticação seja NTLM */
  char* sAuthMech; /* Mecanismo de autenticação utilizado */
  bool bUseTls; /* Define se usa TLS na conexão */
  char* sTlsKeyFile; /* Arquivo com a chave TLS */
  char* sTlsCertFile; /* Arquivo com o certificado TLS */
  char* sTlsTrustFile; /* Arquivo com a lista Autoridades de Certificação */
  char* sTlsRevokeListFile; /* File com a lista de revogação de certificados */
  bool bStartTls; /* Define se o servidor suporta o comando STARTTLS */
  bool bCheckTlsCertificate; /* Define se o certificado do servidor deve ser verificado */

};
```

Interfaces do componente de notificação:

a) Entrada:
   i. const char* – Mensagem de notificação.
b) Saída:
   i. bool(retorno) – sucesso: true, falha ao executar qualquer notificação: false.
   ii. char** – as mensagens de erro serão concatenadas na posição 0.

Interfaces do componente interno para envio de email:

a) Entrada:
   i. able_mail_t* - email a ser enviado
b) Saída:

i. bool (retorno) – sucesso: true, falha: false.
ii. char** – mensagem de erro é colocada na posição 0.

Interfaces dos programas externos de notificação:

a) Entrada:
   i. Dados da notificação a ser emitida. Estas informações são passadas pela linha de comando como argumentos.
b) Saída:
   i. A saída de um notificador vem de dois lugares: saída padrão e retorno. Tudo o que o notificador enviar para a saída padrão será tratado como uma string que representa uma mensagem de problema encontrado e só é utilizada se o valor de retorno confirmar que houve algum problema. O valor de retorno deve ser 0 se o notificador enviou a mensagem com sucesso, 126 ou 127 se não foi possível sequer tentar realizar a notificação ou qualquer outro valor se a tentativa de notificar falhou.

3.5. Monitoramento

Os componentes que realizam a verificação dos serviços são programas independentes que recebem as informações da configuração e devolvem o resultado da verificação. Estes programas podem ser também *scripts* que executam vários outros programas.

O funcionamento de cada verificador é específico, pois depende de como forem implementados.

O *AvailableD* possui um conjunto de verificadores, um para cada serviço. Alguns dos serviços que possuem a verificação implementada são: SMTP, DNS e HTTP.

Para ajudar na generalização do comando de verificação, existem duas variáveis que possuem a identificação de cada servidor, sendo elas $SERVER e $PORT, que serão substituídas pelo endereço IP do servidor e pela porta para conexão ao serviço, respectivamente.

Para desenvolver verificadores, há alguns poucos detalhes que devem ser observados para que funcionem corretamente com o *AvailableD*:

1. **Valores de Retorno**: um verificador deve retornar 0 somente quando o serviço foi verificado e constatada sua disponibilidade. Se o verificador detectar que o serviço está indisponível, deve ser retornado qualquer outro. Os valores de retorno 126 e 127 são reservados para identificar que o serviço não pode ser verificado, logo, nenhuma ação deve ser tomada, pois a situação atual do serviço não é conhecida.
2. **Recursos Utilizados**: por questões de segurança, os verificadores devem ser cautelosamente desenvolvidos para não utilizarem recursos que possam consistir em brechas de segurança, principalmente porque poderão estar sendo executados com as credenciais de usuários privilegiados do sistema.
3. **Mensagens de Erro**: mensagens somente são utilizadas pelo *AvailableD* quando o notificador retornar algum código diferente de 0. Portanto, somente há necessidade de serem emitidas para informar problemas identificados. A maneira de passar estas mensagens para o *AvailableD* é imprimindo-as na saída padrão. Devem ser evitados caracteres especias e mensagens muito longas.

Funcionamento do componente de monitoramento:

1) Obtém comando de verificação do serviço. Vai para o passo 2;
2) Interpreta as variáveis do comando;
3) Executa comando;
4) Termina a execução.

Funcionamento geral dos verificadores de disponibilidade:

1) Verifica conformidade dos parâmetros passados. Se estiver tudo OK com os parâmetros, vai para o passo 2, senão, para o passo 4.
2) Realiza verificação do serviço. Vai para o passo 3.
3) Analisa o resultado da verificação. Se nenhum problema ocorreu, vai para o passo 5, senão, para o passo 4.
4) Imprime mensagem de erro na saída padrão. Vai para o passo 5.
5) Para a execução, retornando o valor 0 se nenhum problema foi encontrado e o serviço está disponível, 126 ou 127, caso não tenha sido possível verificá-lo (parâmetros insuficientes, por exemplo) ou qualquer outro valor, para informar que o serviço está indisponível.

Na figura 15 está representado o funcionamento do componente de monitoramento. A figura 16 mostra o fluxograma geral dos verificadores de serviços.

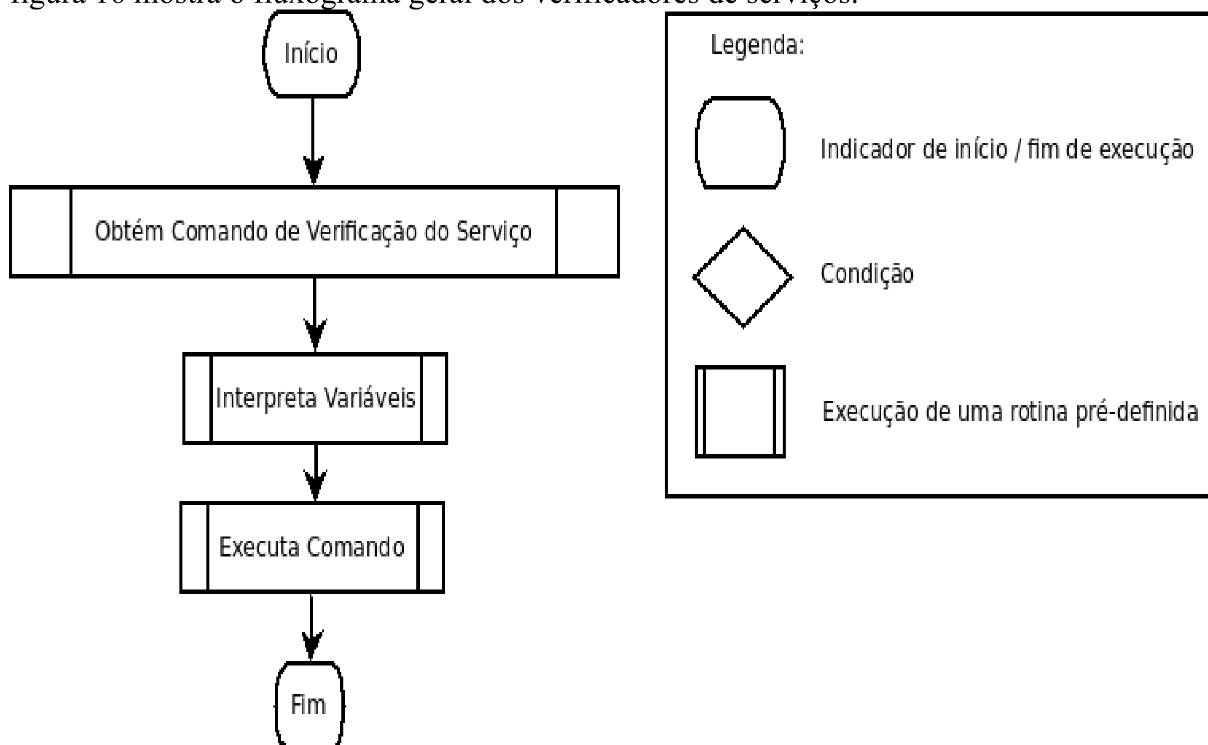


Figura 15: Fluxograma de execução do componente de monitoramento.

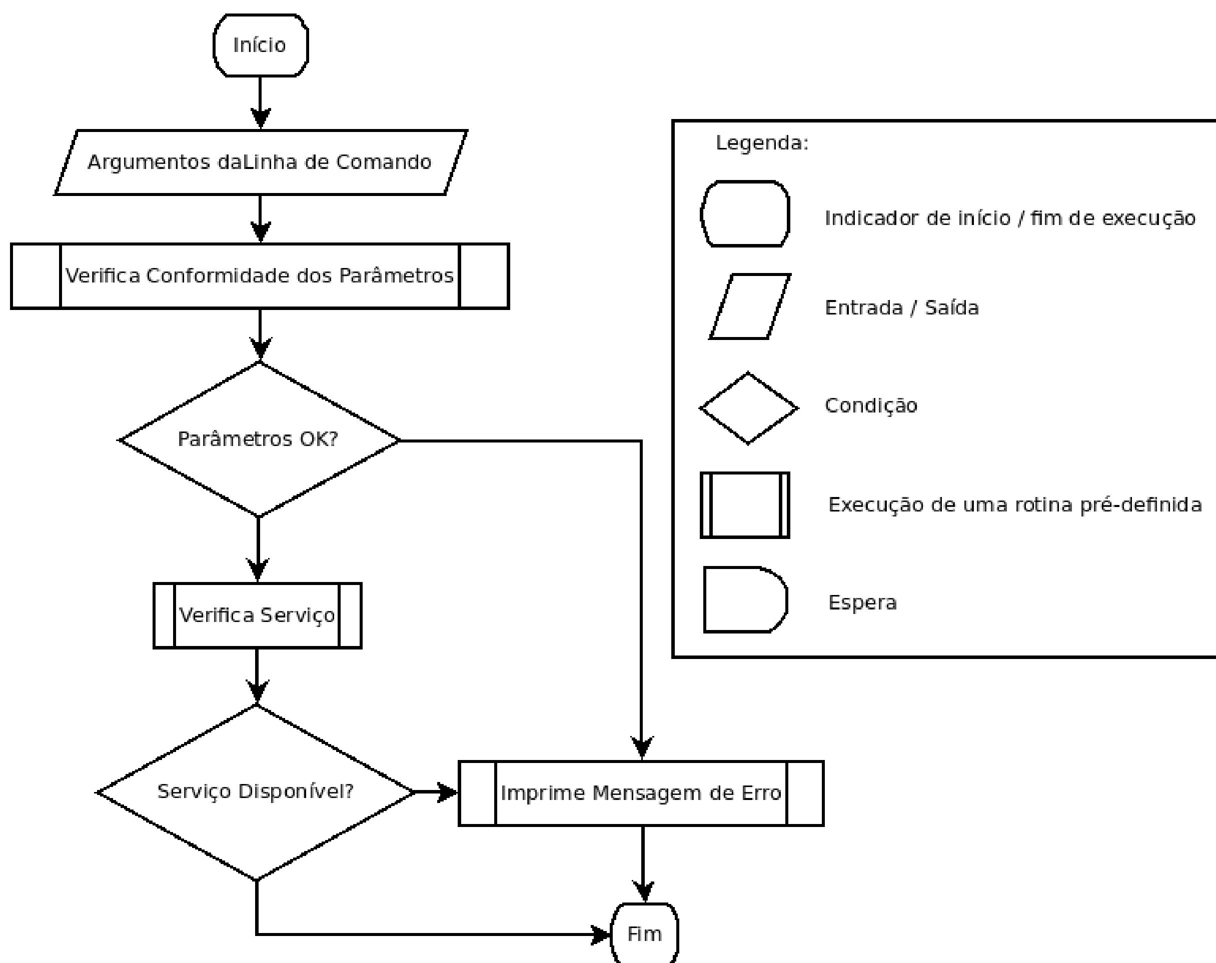


Figura 16: Fluxograma geral de execução dos verificadores de serviços.

Tecnologias para Implementação

Os verificadores deverão ser adaptados dos disponíveis na ferramenta de monitoramento de serviços *Mon*. Seus verificadores estão escritos em PERL.

Estruturas de dados:

Nenhuma nova.

Interfaces dos verificadores:

c) Entrada:
   i. Dados do serviço a ser verificado. Estas informações são passadas pela linha de comando como argumentos.
d) Saída:
   i. A saída de um verificador vem de dois lugares: saída padrão e retorno. Tudo o que o verificador enviar para a saída padrão será tratado como uma string que representa uma mensagem de problema encontrado e só é utilizado se o valor de retorno confirmar que houve algum problema. O valor de retorno deve ser 0 se o serviço estiver disponível, 126 se não foi possível testar o serviço ou qualquer outro valor se o serviço encontra-se indisponível.

3.6. Executor de Ações

Este é o componente responsável por executar as ações definidas em casos de indisponibilidade dos servidores e as ações para quando os mesmos voltarem a estar disponíveis. As regras do *firewall* para redirecionamento de pacotes, portanto, são executadas por este componente.

Com o propósito de prover maior flexibilidade e poder das ações paliativas, é permitido a execução de vários comandos, que não necessariamente sejam manipulação de regras do *firewall*. Se qualquer dos comandos falhar (retornar valor diferente de 0), a ação será considerada não realizada.

Este é o componente mais complexo do sistema, pois é o responsável por tratar da lógica dos redirecionamentos de pacotes.

Em resumo, há uma situação em que não basta apenas adicionar redirecionamento do servidor que se tornou indisponível. Esta situação ocorre quando o servidor é alvo de redirecionamento das requisições de outros servidores que ser tornaram indisponíveis antes. Neste caso, o procedimento a ser realizado é remover estes redirecionamentos, escolher novos servidores substitutos, e realizar os redirecionamentos para eles. Este componente não pode ser executado em paralelo justamente pela situação anteriormente descrita.

Funcionamento do componente:

1) Avalia o resultado da verificação do serviço realizada pelo módulo monitoramento. Se houve alteração na disponibilidade do serviço, vai para o passo 2, senão, para o passo 20.
2) Se o serviço agora encontra-se disponível, vai para 3, senão, vai para 7;
3) Obtém o servidor para o qual o serviço havia sido redirecionado. Vai para o passo 4;
4) Obtém os comandos para remover o redirecionamento realizado. Vai para o passo 5;
5) Interpreta variáveis nos comandos .Vai para o passo 6;
6) Executa os comandos. Vai para o passo 20;
7) Obtém a lista dos servidores que foram redirecionados para o servidor agora indisponível.
8) A lista de servidores já chegou ao fim? Se sim, vai para o passo 16, senão, para o passo 9.
9) Obtém servidor da lista. Vai para o passo 10;
10) Obtém comandos para remover o redirecionamento do servidor. Vai para o passo 11;
11) Interpreta variáveis dos comandos e os executa. Vai para o passo 12;
12) Escolhe um novo servidor substituto. Vai para o passo 13;
13) Obtém os comandos para adicionar novo redirecionamento. Vai para o passo 14;
14) Interpreta variáveis dos comandos e os executa. Vai para o passo 15;
15) Avança a lista de servidores. Vai para o passo 8;
16) Escolhe um servidor substituto para o servidor agora indisponível. Vai para o passo 17;
17) Obtém os comandos para adicionar o redirecionamento. Vai para o passo 18;
18) Interpreta variáveis dos comandos. Vai para o passo 19;
19) Executa os comandos. Vai para o passo 20;
20) Termina a execução.

A figura 17 mostra o fluxograma do componente executor de ações.

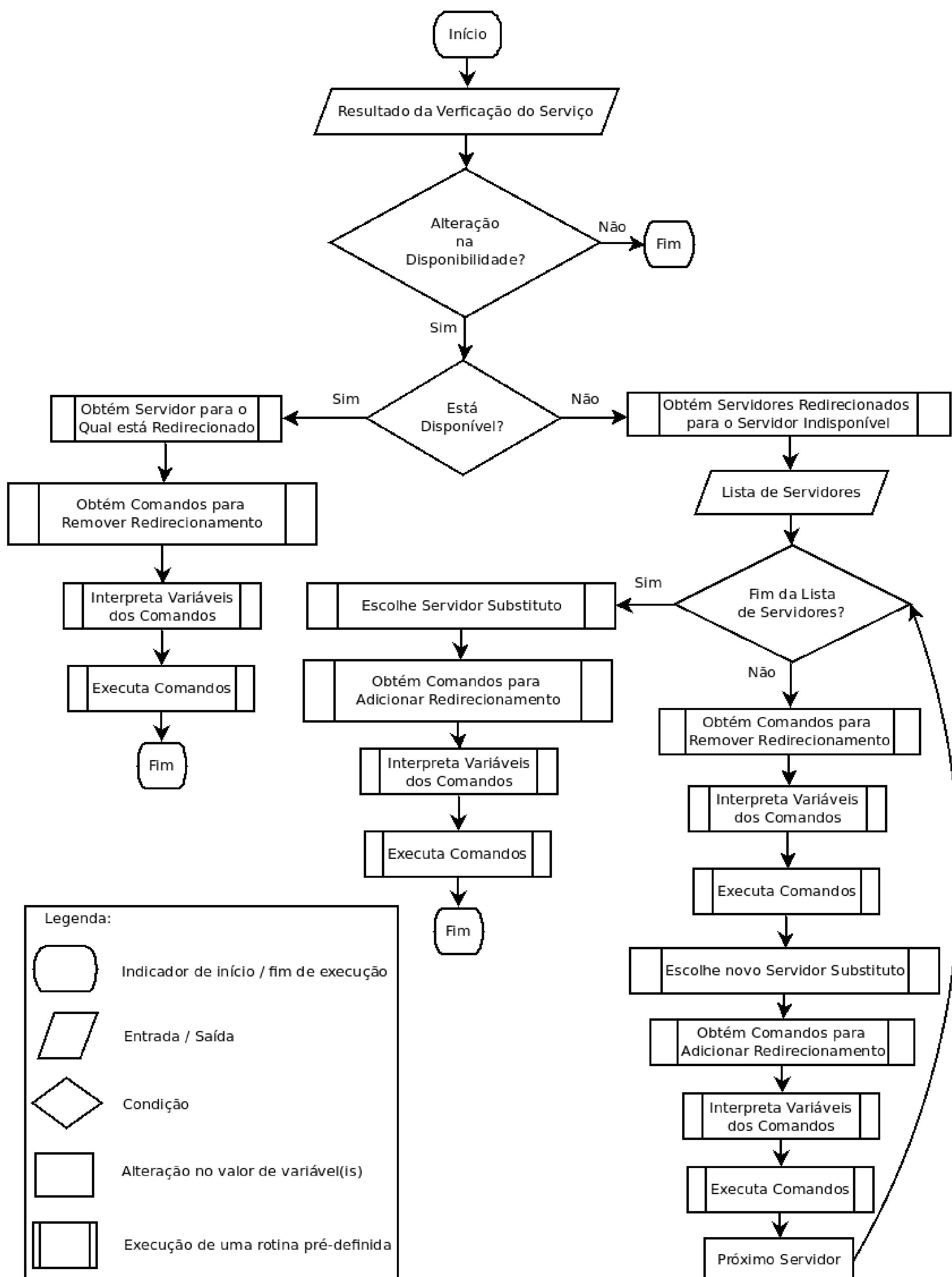


Figura 17: Fluxograma de execução do componente executor de ações.

Tecnologias para Implementação

Os comandos de *firewall* a serem executados deverão retornar 0 em caso de sucesso. Caso o *firewall* utilizado retorne algum outro valor, deverá ser desenvolvido um *script* que faz a adaptação necessária de valores de retorno. O f*irewall* mais comumente utilizado em ambientes GNU/Linux, o IPTables, por exemplo, retorna 0 somente quando seu comando foi corretamente executado, ou seja, já está em conformidade para uso pelo sistema;

Estruturas de dados:
Nenhuma nova.

Interfaces do componente de execução de ações:

c) Entrada:
   i. able_list_t*(char*) - Lista de comandos de ação a serem executados.

d) Saída:
   i. bool(retorno) – sucesso: true, falha ao executar algum dos comandos: false.
   ii. char** – as mensagens de erro serão concatenadas na posição 0.

3.7. Integração de Componentes

Este tópico trata a lógica de integração dos componentes, ou seja, a lógica responsável por ligar os componentes de forma a atuarem como um único software (o *AvailableD*) que atenda aos requisitos e cumpra a especificação funcional definida.

Com o objetivo de aproximar mais da programação, a integração será demonstrada em pseudo código.

O pseudo código descrito abaixo não tem a intenção de ser fiel as estruturas de dados e interfaces anteriormente definidas e sim de abstrair a lógica de implementação, mesmo que para isso alguns detalhes sejam omitidos.

```
CONFIGURACAO:= carregaConfiguracao(ARQUIVO_CONFIGURACAO, &MENSAGEM);
se nao CONFIGURACAO entao:

 mostra(MENSAGEM);
 sair();

fim se

para cada CONFIGURACAO->SERVICOS como SERVICO faca:

 para cada SERVICO->SERVIDORES como SERVIDOR faca:

  thread:

   para SEMPRE faça:

    RESULTADO := monitoraServiço(SERVIDOR->CONFIGURACAO, &MSGVERIF);

    mutexLock();
    se RESULTADO é INDISPONÍVEL e SERVIDOR->MUDOU_SITUACAO entao: // Estava disponível mas não está mais

     SERVIDOR_SUBSTITUTO := escolheServidorSubstituto(SERVIDOR->CONFIGURACAO);
     adicionaRedirecionamento(SERVIDOR, SERVIDOR_SUBSTITUTO, &MSGACAO);
     log(MSGACAO);
     thread:
      notifica(uneMsgs(MSGVERIF, MSGACAO));
     fim thread

    senao se  RESULTADO é  DISPONIVEL e nao SERVIDOR->MUDOU_SITUACAO  entao: // Serviço voltou a estar disponível

     removeRedirecionamento(SERVIDOR,  &MSGACAO);
```

```
      log(MSGACAO);
      thread:
        notifica(uneMsgs(MSGVERIF, MSGACAO));
      fim thread

    fim se
    mutexUnlock();

    esperaProximaVerificacao(SERVIDOR->CONFIGURACAO);

   fim para

  fim thread

 fim para cada

fim para cada

viraDaemon();
dormeParaSempre();
```

4. Padrões de Codificação

A fim de manter um código legível e de fácil manutenção, faz-se necessário determinação de alguns padrões norteadores do desenvolvimento de código. A seguir, são apresentados os padrões definidos para codificação do *AvailableD*.

4.1. Nomes e Declaração de Variáveis

Os nomes de variáveis seguem o seguinte padrão:

1) A declaração do tipo da variável deve ficar na mesma linha de seu nome. Ex: int i;
2) Primeira letra de cada palavra maiúscula;
3) Sem uso de "_" no nome;
4) Nomes auto explicativos e em inglês;
5) Para mais fácil identificação dos tipos das variáveis, os primeiros caracteres do nome (prefixo), em minúsculo, devem representar o tipo da variável, de acordo com a seguinte tabela:

Tipo	Inicial	Exemplo
inteiro, long	i	iCount
float, double	n	nAverage
char, char[], char*	s	sString
estrutura	o	oStack
ponteiro	Tipo conteúdo + p	opStack, spName
função	f	fCallback

Tabela 1: Prefixos para nomes de variáveis

4.2. Nomes e Declaração de Funções

As funções do *AvailableD* seguem o seguinte padrão:

1) Iniciam pelo prefixo "able_". Isso facilita a identificação de funções externas;
2) A declaração do tipo de retorno deve ficar na mesma linha do nome da função;

3) Nomes em inglês;
4) Sem uso de "_", exeto no prefixo;
5) A primeira letra do nome deve ser minúscula. As demais em maiúscula. Ex: able_functionName;
6) O nome da função deve explicar o que a mesma faz. Ex: able_runShellCommand;
7) Parâmetros separados por vírgula mais 1 espaço em branco. Ex: able_func(int i1, int i2);
8) Não deve haver espaço entre nome da função e abre parênteses. Nos demais detalhes, segue o mesmo padrão dos elementos de controle de fluxo. Ex: able_func();

4.3. Comando de controle de fluxo

A forma de apresentação dos comandos de controle de fluxo são de grande influência na legibilidade do código, pois os mesmos dividem o código em blocos. Sendo assim, devem ser seguidos os seguintes padrões:

1) Após o nome do comando, deve ter um espaço em branco e então o abre parênteses. Ex: if (;
2) Não deve haver espaço entre os parênteses e seu conteúdo. Ex: if (condition);
3) Após o fecha parênteses deve ter um espaço em branco e então o abre chaves. Ex: while (condition) {;
4) O fecha chaves deve ser alinhado com a primeira letra do nome do comando. Ex:
```
switch (i) {
}
```
5) Sempre que houver mais de uma linha de código interna ao comando, deve ser deixada uma linha em branco antes e depois do código interno. Ex:
```
for (i = 0; i < 10; i++) {

  declaração1;
  declaração2;

}
```
6) Quando o código dentro dos parênteses for muito extenso, pode ser quebrado e alinhado com o início do código. Quando a quebra for em condições, o operador condicional deve fazer parte da quebra. Ex:
```
if (condicao1 && condicao2 && condicao3
    && condicao4, && condicao5, && condicao6) {
```
7) No caso específico do comando for, quando o conteúdo entre os parênteses for muito extenso, a quebra deve seguir o exemplo:
```
for (expressa_inicio;
      condicao;
     expressao_frim) {
```

4.4. Chamadas de funções

As chamadas de funções constituem grande parte do código. Para melhor legibilidade das chamadas devem seguir os seguintes padrões:

1) Sem espaço entre nome da função e abre parênteses. Ex: able_func(;

2) Sem espaço entre parênteses e código interno. Ex: able_func(i1);
3) Argumentos espaçados por vírgula mais 1 espaço em branco. Ex: able_func(i1, i2, i3, i4);
4) Quando o código interno aos parênteses for muito extenso, realiza-se a quebra a partir do segundo parâmetro, alinhando sempre com o início do primeiro. O fecha parênteses deverá ser allinhado com o abre parênteses. Ex:

```
able_func(i1, i2,
          i3, i4
         );
```

4.5. Operadores

Padrões:

1) Um espaço em branco entre operadores e operandos. Ex: i1 + i2;
2) Os operadores unários, fazem exceção à regra anterior, pois devem ficar colados aos operandos. Ex: i++;
3) O operador de atribuição e suas variantes devem ficar alinhados pelo símbolo '=' quando em linhas consecutivas de código são realizadas atribuições. Ex:

```
iSize      = 20;
iCount    += 2;
sSeparator = ',';
oStack     = able_newStack();
```

4.6. Identação

A identação é um dos aspectos mais importantes para a legibilidade do fonte. Para o *AvailableD*, os padrões de identação são:

1) Sem utilização de TAB;
2) Usar 2 espaços em branco para cada nível de identação;
3) Variáveis globais, definições e funções devem ficar sempre no primeiro nível de identação;
4) A indentação (os 2 espaços) é aplicada ao código interno de cada novo bloco.

4.7. Comentários e Documentação do Código

Padrões:

1) Os comentário devem sempre ser do tipo /* */;
2) Para documentação deve ser utilizado o padrão *Javadoc* de comentários, que é suportado por muitas ferramentas geradoras de documentação;
3) Em inglês.

4.8. Demais padrões

1) Tudo deverá ser escrito e documentado em inglês, pois é um idioma compreendido pela maioria dos desenvolvedores. É praticamente o idioma padrão para desenvolvimento de software;
2) Cada linha deve conter no máximo uma instrução. Não deve ser feito algo do tipo: i1 = 1; i2 = 8;

3) O limite máximo para cada linha de código deve ser de 120 colunas. Isso garante uma melhor visualização em máquinas com menor resolução de tela;

Referências

1. Documento de requisitos do Methodology Explorer, disponível em: http://www.cin.ufpe.br/~mexplorer/metodologia/requisitos/documentoRequisitos.doc, acessado em 10/05/2011.
2. PRESSMAN, Roger S. *Engenharia de Software.* McGraw-Hill, 2006.
3. SOMMERVILLE, Ian. *Software Engineering*. 8ª ed. Addison Wesley, 2006.
4. WIKIPEDIA. Engenharia de Requisitos. [Online]. Disponível em: http://pt.wikipedia.org/wiki/Engenharia_de_requisitos, acessado em 15/06/2011
5. The netfilter.org "iptables" project. [Online]. Disponível em: http://www.netfilter.org/projects/iptables/, acessado em 01/09/2011
6. WIKIPEDIA. Javadoc. [Online]. Disponível em: http://en.wikipedia.org/wiki/Javadoc, acessado em 01/10/2011
7. Linux-HA. [Online]. Disponível em: http://www.linux-ha.org, acessado em 02/11/2011
8. Linux Virtual Server. [Online]. Disponível em: http://www.linuxvirtualserver.org/, acessado em 02/10/2011
9. Nagios. [Online]. Disponível em: http://www.nagios.org/, acessado em 02/10/2011
10. Flex: The Fast Lexical Analyzer. [Online]. Disponível em: http://flex.sourceforge.net/, acessado em 05/10/2011
11. MSMTP. [Online]. Disponível em: http://msmtp.sourceforge.net/, acessado em 06/11/2011
12. The GNU Transport Layer Security Library. [Online]. Disponível em: http://www.gnu.org/s/gnutls/, acessado em 06/10/2011
13. The OpenSSL Project. [Online]. Disponível em: http://www.openssl.org/, acessado em 06/10/2011
14. GNU General Public License. [Online]. Disponível em: http://www.gnu.org/copyleft/gpl.html, acessado em 07/10/2011
15. Mon - Service Monitoring Daemon. [Online]. Disponível em: https://mon.wiki.kernel.org/, acessado em 08/10/2011
16. Doxygen Documentation. [Online]. Disponível em: http://doxygen.org, acessado em 01/12/2011

APÊNDICE H – Scripts utilizados para os testes

Figura H.1 – Código do *script* para coleta da utilização da UCP

```
#!/bin/bash
cputime2sec() {
  filecontent=$1
  for line in $filecontent; do
    ismin=1
    secs=0
    IFS=":"
    for field in $line; do
      if [ $ismin -eq 1 ]; then
        secs=`echo "$field * 60" | bc`
        ismin=0
      else
        secs=`echo "$field + $secs" | bc`
        echo $secs
      fi
    done
    IFS="\n"
  done
}
secdiff() {
  filecontent=$1
  prevsec="-1"
  for line in $filecontent; do
    # Get the first sec
    if [[ "$prevsec" = "-1" ]]; then
      prevsec=$line
      continue;
    fi
    secsdiff=`echo "$line - $prevsec" | bc`
    if [[ $secsdiff =~ ^\. ]]; then
      echo "0$secsdiff"
    else
      echo $secsdiff
    fi
    prevsec=$line
  done
}
multiplyby() {
  multiplier=$1
  filecontent=$2
  for line in $filecontent; do
    result=`echo "$line * $multiplier" | bc`
    if [[ $result =~ ^\. ]]; then
      echo "0$result"
    else
      echo $result
    fi
  done
}
divideby() {
  divisor=$1
  filecontent=$2
  for line in $filecontent; do
    result=`echo "scale=2; $line / $divisor" | bc`
    if [[ $result =~ ^\. ]]; then
      echo "0$result"
    else
      echo $result
    fi
  done
}
if [ $# -lt 1 ]; then
  echo "Usage: cpu-usage <gathering_number> [cpus_number] [gathering_frequency]"
  exit 1;
fi;
gathering_number=$1; cpus_number=1;
gathering_frequency=1.0;
if [ $# -gt 1 ]; then
  cpus_number=$2
fi;
if [ $# -gt 2 ]; then
  gathering_frequency=$3
fi;
out=`top -b -d $gathering_frequency -n $gathering_number`
out=`echo "$out" | grep 'availabled *start *$' | awk '{print $5}' | sed '/0:00.0/d'`
out=`cputime2sec "$out"`; out=`secdiff "$out"`; out=`divideby $gathering_frequency "$out"`
out=`multiplyby 100 "$out"`; out=`divideby $cpus_number "$out"`; echo "$out"; exit 0
```

Fonte: do autor

Figura H.2 – Código do *script* para coleta da memória virtual alocada

```
#!/bin/bash
tomegabyte() {
  filecontent=$1
  for line in $filecontent; do
    if [[ "$line" = "-1" ]]; then
      echo "-1"
      continue
    elif [[ $line =~ g$ ]]; then
      tmp=`echo $line | sed s/g//g`
      tmp=`echo "scale=2; $tmp * 1024" | bc`
      if [[ $tmp =~ ^\. ]]; then
        echo "0$tmp"
      else
        echo $tmp
      fi
    elif [[ $line =~ m$ ]]; then
      echo $line | sed s/m//g
    else
      tmp=`echo "scale=2; $line / 1024" | bc`
      if [[ $tmp =~ ^\. ]]; then
        echo "0$tmp"
      else
        echo $tmp
      fi
    fi
  done
}
sumsep() {
  filecontent=$1
  sum=0
  first=1
  for line in $filecontent; do
    if [ $first -eq 1 ]; then # Ignores first, if is separator
      first=0
      if [[ "$line" = "-1" ]]; then
        continue;
      fi
    fi
    if [[ "$line" = "-1" ]]; then
      if [[ $tmp =~ ^\. ]]; then
        echo "0$sum"
      else
        echo $sum
      fi
      sum=0
      continue
    else
      sum=`echo "scale=2; $sum + $line" | bc`
    fi
  done
}

if [ $# -lt 1 ]; then
  echo "Usage: mem-virt-alloc <gathering_number> [gathering_frequency]"
  exit 1;
fi;
gathering_number=$1
gathering_frequency=1.0
if [ $# -gt 1 ]; then
  gathering_frequency=$2
fi;

out=`top -b -d $gathering_frequency -n $gathering_number`
out=`echo "$out" | grep '\(VIRT\)\|\(\.monitor \)\|\(availabled *start *\)$' \
     | awk '{print $2}' | sed '/0:00.0/d' | sed 's/.*VIRT.*/-1/g'`
out=`tomegabyte "$out"`
out=`sumsep "$out"`
echo "$out"
exit 0;
```

Fonte: do autor

Figura H.3 – Código do *script* para coleta do tráfego de rede gerado

```
#!/bin/bash
# This script gets bytes from network interface. If the interface is used to other things
# than perform service checks, this script is not useful
#
# Requires bytetraf version provided with this script

if [ $# -lt 2 ]; then
  echo "Usage: net-usage <gathering_number> <interface> [{gathering_frequency}s|h|d]"
  exit 1
fi

gathering_number=$1
interface=$2
gathering_frequency=1s
if [ $# -gt 2 ]; then
  gathering_frequency=$3
fi

# Get data from bytetraf
out=`./bytetraf $gathering_frequency $interface $gathering_number`

# Extract columns recv and tran in bytes, and sum them
out=`echo "$out" | awk '{print $5, $8}' | sed 's/+\|\[\|\]//g' | awk '{print $1 + $2}'`

# Prints
echo "$out"
exit 0
```

Fonte: do autor

Figura H.4 – Código *script* para simulação de falhas em serviços

```
# Specific for services: bind9, apache2 and postfix.
echo $$
if [ $# -lt 2 ]; then
  echo "Usage: service_up_down.sh <sleep_time_up> <sleep_time_down> [num_outages]"
  exit 1
fi
startstamp=`date "+%s"`
start_stop="stop"
dns_downtime=0; smtp_downtime=0; http_downtime=0; dns_time=0; smtp_time=0; http_time=0;

do_exit() {
  if [ "$start_stop" == "start" ]; then
    service bind9 start
    final_time=`date "+%s"`
    dns_downtime=`expr $dns_downtime + $final_time - $dns_time`
    service postfix start
    final_time=`date "+%s"`
    smtp_downtime=`expr $smtp_downtime + $final_time - $smtp_time`
    service apache2 start
    http_downtime=`expr $http_downtime + $final_time - $http_time`
    final_time=`date "+%s"`
  fi
  echo "$dns_downtime $smtp_downtime $http_downtime"
  finalstamp=`date "+%s"`
  expr $finalstamp - $startstamp;
  exit 0;
}

trap 'do_exit' 1 2
sleep_time_up=$1
sleep_time_down=$2
[ $3 ] && num_downtimes=$3 || num_downtimes=-1

while [ true ]; do

  service bind9 $start_stop
  final_time=`date "+%s"`
  if [ "$start_stop" == "start" ]; then
    dns_downtime=`expr $dns_downtime + $final_time - $dns_time`
  else
    dns_time=`date "+%s"`
  fi

  service postfix $start_stop
  final_time=`date "+%s"`
  if [ "$start_stop" == "start" ]; then
    smtp_downtime=`expr $smtp_downtime + $final_time - $smtp_time`
  else
    smtp_time=`date "+%s"`
  fi

  service apache2 $start_stop
  final_time=`date "+%s"`
  if [ "$start_stop" == "start" ]; then
    http_downtime=`expr $http_downtime + $final_time - $http_time`
  else
    http_time=`date "+%s"`
  fi

  if [[ "$start_stop" == "start" ]]; then # End of outage

    start_stop="stop"
    if [ $num_downtimes -gt -1 ]; then # Downtimes defined by user
      num_downtimes=$(( $num_downtimes - 1 ))
      if [ $num_downtimes -lt 1 ]; then
        do_exit
      fi
    fi
    sleep $sleep_time_up

  else # Downtime started
    start_stop="start"
    sleep $sleep_time_down
  fi

done
```

Fonte: do autor

Figura H.5 – *Script* PERL que abre conexão via *socket*

```
#!/usr/bin/perl
# Returns 0 if open socket, 1 if not. Returns 127 case insuficient arguments

use Socket;
use Sys::Hostname;

my ($server, $port, $timeout) = @ARGV;

if ($port == undef) {

  print "Usage: socket-test SERVER PORT [TIMEOUT]. TIMEOUT default is 5 (seconds).";
  exit 127;

}

if ($timeout == undef) {
  $timeout = 5;
}

# Alarm timeout
local $SIG{ALRM} = sub { exit 1; };
alarm $timeout;
# Open socket
if (!&openSocket($server, $port)) { # Failure
  exit 1;
}
exit 0;

sub openSocket {

  my ($host, $port) = @_;
  my $proto = (getprotobyname('tcp'))[2];

  if (!defined $proto) {
    return 0;
  }

  $host = (gethostbyname($host))[4];
  if (!defined $host) {
    return 0;
  }

  my $that = sockaddr_in($port, $host);

  if (!socket (S, &PF_INET, &SOCK_STREAM, $proto)) {
    return 0;
  }

  if (!connect(S, $that)) {
    return 0;
  }

  select(S);
  $| = 1; # Forces flush after every write or print.
  select(STDOUT);

  return 1;

}
```

Fonte: do autor

Figura H.6 – *Script* para coleta dos *outages* dos serviços HTTP e SMTP

```
if [ $# -lt 2 ]; then
  echo "Bad usage"
  exit 127
fi

server=$1
port=$2
if [ $# -gt 2 ]; then
  timeout=$3
else
  timeout=1
fi

do_exit() {
  if [ $status -eq 0 ]; then
    backstamp=‘date "+%s"‘
    downtime=‘expr $backstamp - $failstamp‘
    downtimetot=‘expr $downtimetot + $downtime‘
    echo $downtime
  fi
  echo $downtimetot
  finalstamp=‘date "+%s"‘
  expr $finalstamp - $startstamp
  exit 0
}

trap ’do_exit’ 1 2
startstamp=‘date "+%s"‘
downtime=0
downtimetot=0
teststamp=0
failstamp=0
backstamp=0
status=1 # 0 - Unavailable. 1 - Available.
while [[ true ]]; do

  teststamp=‘date "+%s"‘
  ./socket-test $server $port $timeout
  if [[ $? -eq 0 ]];  then # Available
    if [ $status -eq 0 ]; then
      backstamp=$teststamp
      downtime=‘expr $backstamp - $failstamp‘
      downtimetot=‘expr $downtimetot + $downtime‘
      echo $downtime
      status=1
    fi
    sleep 1

  else # Unavailable
    if [ $status -eq 1 ]; then
      failstamp=$teststamp
      status=0
    fi
    sleep 1

  fi

done
```

Fonte: do autor

Figura H.7 – *Script* para coleta dos *outages* do serviço DNS

```
#!/bin/bash
do_exit() {
  if [ $status -eq 0 ]; then
    backstamp=`date "+%s"`
    downtime=`expr $backstamp - $failstamp`
    downtimetot=`expr $downtimetot + $downtime`
    echo $downtime
  fi
  echo $downtimetot
  finalstamp=`date "+%s"`
  expr $finalstamp - $startstamp
  exit 0
}

trap 'do_exit' 1 2
startstamp=`date "+%s"`
downtime=0
downtimetot=0
teststamp=0
failstamp=0
backstamp=0
status=1 # 0 - Unavailable. 1 - Available.
while [[ true ]]; do

  teststamp=`date "+%s"`
  out=`nslookup -timeout=1 -retry=0 pingo.ginete.tche 10.10.10.1`

  if [[ $out =~ 10.10.10.13 ]]; then # Available
    if [ $status -eq 0 ]; then
      backstamp=$teststamp
      downtime=`expr $backstamp - $failstamp`
      downtimetot=`expr $downtimetot + $downtime`
      echo $downtime
      status=1
    fi
    sleep 1

  else # Unavailable
    if [ $status -eq 1 ]; then
      failstamp=$teststamp
      status=0
    fi

  fi

done
```

Fonte: do autor

APÊNDICE I – Exemplo de configuração do AvailableD utilizada nos testes

Figura I.1 – Configuração do AvailableD para monitorar 6 réplicas

```
# Informações padrões globais para os serviços
check_timeout = 2s;
wait_time = 5s;
log_facilities = file;
log_file = /var/log/availabled.log;
check_path = /usr/lib/availabled/plugins/check;

action_commands = "iptables -t nat -A PREROUTING -p tcp -d $BROKEN_SERVER \
                   --dport $BROKEN_PORT -j DNAT --to $ACTIVE_SERVER:$ACTIVE_PORT";

undo_action_commands = "iptables -t nat -D PREROUTING -p tcp -d $BROKEN_SERVER \
                        --dport $BROKEN_PORT -j DNAT --to $ACTIVE_SERVER:$ACTIVE_PORT";

service dns {

  # Informação padrão para o serviço dns
  check_command = "dns-query.monitor --name \"ginete.tche\" $SERVER";

  action_commands = "iptables -t nat -A PREROUTING -p udp -d $BROKEN_SERVER \
                    --dport $BROKEN_PORT -j DNAT --to $ACTIVE_SERVER:$ACTIVE_PORT";

  undo_action_commands = "iptables -t nat -D PREROUTING -p udp -d $BROKEN_SERVER \
                         --dport $BROKEN_PORT -j DNAT --to $ACTIVE_SERVER:$ACTIVE_PORT";

  server 10.10.10.1:53 {
    id = 1;
    redirect_to = 2;
  }

  server 10.10.10.2:53 {
    id = 2;
    redirect_to = 1;
  }

}

service smtp {

  check_command = "smtp.monitor --port $PORT --helo-domain \"availabled.com\" $SERVER";

  server 10.10.10.1:25 {
    id = 1;
    redirect_to = 2;
  }

  server 10.10.10.2:25 {
    id = 2;
    redirect_to = 1;
  }

}

service http {

  check_command = "http.monitor --path \"index.html\" --port $PORT $SERVER";

  server 10.10.10.1:80 {
    id = 1;
    redirect_to = 2;
  }

  server 10.10.10.2:80 {
    id = 2;
    redirect_to = 1;
  }

}
```

Fonte: do autor